Université d'Ottawa · University of Ottawa

Identification of Moving Objects in Colour Digital Video Sequences

by

Gaétan Noël

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of

M.A.Sc. in Electrical and Computer Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering
School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

À grand-maman et grand-papa,
pour qui mes souvenirs
sont inoubliables.

Table of Contents

List of Figures

List of Equations

List of Tables

List of Code Samples and Algorithms

Abstract

In the past decade, there has been a tremendous increase in the use of digital video through computer systems. However, in many applications, the video size and frame rate suffer from the fact that enormous amounts of bandwidth and storage space are still required by digital video. Nevertheless, home and industry level computers have become powerful enough to decompress and display good quality video through the use of specialized hardware such as video capture boards and video display cards without using up all CPU cycles: we now have the ability to process video in real-time.

Research on video processing is still at an early stage but is now attracting attention. Concepts such as motion detection, object tracking and object recognition can be very useful in many applications and have become more feasible. Another area that could benefit from new video processing algorithms is video indexing in multimedia database systems. Compression of video allows it to be stored in vast quantity, but the problem is the retrieval of video that has been stored. Multiple techniques can be used to accelerate the video search process but most of them do not help in understanding the contents of the video, which often must be done manually.

In this thesis, a moving objects identification algorithm called *SmartID* has been designed. It brings some level of improvement in many sectors of computerized video, but in particular in the three sectors mentioned above: video compression, video processing applications and video indexing. This research also encompassed the design and implementation of a digital video surveillance application that demonstrates the usefulness of the algorithm.

Acknowledgements

I would like to express my appreciation to Dr. Ahmed Karmouch who has been my thesis supervisor during my M.A.Sc. program. Dr. Karmouch has always been supportive of me when the reorganizations of the companies I have worked with were taking place.

I would like to thank NSERC (Natural Sciences and Engineering Research Council of Canada) and CITO (Communications and Information Technology Ontario) who both made this research financially possible.

Thanks to Robert Menard, I was given the chance to join this cooperative project between Televitesse Systems and the University of Ottawa. Televitesse Systems was later sold to Telexis Corporation, with which I pursued the work. In its turn, Telexis Corporation became March Networks Corporation, with which I completed my project. Thanks to these three companies for supporting me.

Special thanks go to the MAP team at March Networks, and more specifically to Roger Maclean, Paul Streatch, Michael Baynger and Sean McDowell for their help during my M.A.Sc. program.

For their encouragements and support, I am truly grateful to my parents. Without them, this project never would have started.

Finally, I would like to convey my very special thanks to my wife, Vickie, for her patience and constant support, and to my son, Étienne, who gave me great support just by his presence from the day he was born, two years ago.

Chapter 1

Introduction

1.1 Motivation

The recent advances in multimedia and digital hardware technologies have made possible the transmission of digital video streams over a variety of networks (e.g. POTS, ISDN and ATM), thus leading to a greater need for information logging and retrieval. Applications that are using digital video transmission and storage range from teleconferencing and interactive television to security surveillance.

In the early days of database systems, text was the only medium that needed to be dealt with, so there was no real storage capacity and information retrieval problem. With

the arrival of digital video and audio, storage and bandwidth requirements have increased tremendously and information retrieval remains a challenge. The important features of multimedia database systems must include efficient compression techniques to reduce the storage requirements, as well as effective indexing to allow information to be searched for and retrieved as quickly as possible.

Uncompressed video streams normally require very large amounts of bandwidth for transmission and storage; therefore the use of compression techniques such as MPEG, JPEG, H.261 and H.263 has become a requirement rather than a feature. As a general rule, video compression techniques are geared towards compression time and ratios and do not take into consideration that the video might have to be scanned. This makes it difficult to index video images in a database. It is one thing to be able to log and transmit video efficiently, but another to actually know what the content of the video represents.

Different approaches have been taken to try to describe the contents of video, and most of these fall in one of two categories. The first one is to generate time-based keywords and attributes describing the video by actually looking at its content (most often done manually). The second one is to analyse and understand the video through computerized techniques such as object identification and recognition. Both video compression and indexing are time-consuming tasks and in multimedia systems, the work is more than often duplicated when they are not done in conjunction. There is much to be gained by unifying these elaborate tasks.

In addition to efficient storage and retrieval, a number of video applications could benefit from identifying and extracting moving objects from video sequences. These currently are mainly surveillance related but there are many other uses for them (e.g. multiple level compression, remote sensing, teleconferencing, event triggering and recognition). The identification of moving objects is usually simply called object identification. Motion detection is used to detect temporal motion in a video sequence without necessarily knowing the relationship between the detected motion and the possible objects that are moving. On the other hand, object identification is used to determine which objects in a sequence are moving and where they are located. At a higher level, the tracking of the moving objects could be performed by an object tracking algorithm.

In this thesis, we have developed an algorithm that performs moving objects identification and have demonstrated the ability of this algorithm to execute in a real-time environment by developing a software application that uses the algorithm.

1.2 Objectives

The main objective of this thesis is to develop a moving objects identification algorithm, *SmartID*, which can identify any kind of moving objects in digital video sequences. To demonstrate the benefits of using *SmartID* in an application, the thesis also covers the design and implementation of a Digital Video Surveillance application

named DVS. DVS identifies moving objects by using *SmartID*, efficiently archives video through its own database system and allows search and play back of archived video.

Typical video surveillance applications record full video sequences even when there are no objects moving in them, thus using a large amount of storage space for very little useful information. Furthermore, these applications normally allow a user to search and play back video based on time only, which makes it difficult to find objects that are moving in a large video database. Consequently, this thesis presents a way to facilitate the storage and retrieval of moving objects, so that end users can retrieve the information that they are looking for in just a few steps. This demonstrates the ability of *SmartID* to serve as a video compression and indexing aid.

1.3 March Networks Partnership

Throughout the research presented in this thesis, March Networks has played a key role in supplying the programming and hardware infrastructures and the general video knowledge that was required to successfully accomplish the work. The partnership started between Televitesse systems and the University of Ottawa, but Televitesse was eventually sold to Telexis Corporation. At that point the research was reoriented to better suit the needs of the company. In its turn, Telexis Corporation became March Networks Corporation in the summer of 2000, but that did not affect the research.

Televitesse had a great programming platform for the development of multimedia applications, which was enhanced by Telexis with the support for specialized video hardware that can stream video over networks. All the multimedia technology has been transferred to March Networks, and is used as the development base for the research described in this thesis.

1.4 Main contribution of the thesis

The main contribution of this thesis is the design and implementation of a moving objects identification algorithm called *SmartID* that is independent of any *a priori* information about a scene in which it would perform and about the moving objects it would identify. The algorithm does not rely on a database of known objects or object types, implying that it does not require any kind of training to be carried out before it can be used. *SmartID* can detect all kinds of moving objects, rigid, non-rigid and freeform, even when they temporarily become stationary. In the case where objects become permanently stationary, they stop being detected as moving objects.

This algorithm's design and implementation have been done in such a way that the algorithm can be used in real-time systems, implying that processing power requirements have been kept as low as possible.

Furthermore, this research also led to the design and implementation of a video surveillance application that can use a remote video camera as its video source to be

processed by the *SmartID* algorithm. The processed video can be stored and retrieved efficiently through a video indexing database that also has been developed for this thesis.

1.5 Thesis outline

In chapter 2, a brief description of the kind of objects we are interested in identifying is given, followed by some theory on video decomposition and reconstruction. An overview of the concepts related to the identification of moving objects is also given.

In chapter 3, we explain in detail the moving objects identification algorithm, named *SmartID*, that is developed in this thesis. This chapter also covers briefly the description of objects after their identification and extraction from the video sequences.

In chapter 4, we cover the design of DVS, the application that demonstrates the usefulness of *SmartID* in a few areas of video manipulation (e.g. storage, retrieval, compression and identification).

In chapter 5, we discuss the implementation of DVS, explain how the application interfaces with March Networks' code base, give details on the architecture of the video bus in the application and also summarize the functionality of each video module.

In chapter 6 we provide a discussion on the performance and efficiency of the developed algorithm and application. Finally, we give a summary of the thesis and propose suggestions for future research directions in chapter 7.

Chapter 2

Object Identification

2.1 Introduction

In order to improve video surveillance applications at March Networks, the development of a video processing algorithm that would perform moving object identification was required. This led to the development of a Digital Video Surveillance application called DVS. This application would need to have the capability of highlighting moving objects in digital video and also of storing and retrieving the video. For that purpose, a moving objects identification algorithm called *SmartID* was developed. Before we discuss *SmartID* any further, a good understanding of the theory behind the identification of moving objects and the concepts related to it is necessary.

This chapter first defines what the objects of interest are for our purposes. It then provides theory regarding video decomposition and reconstruction and gives an overview of concepts and techniques related to the identification of moving objects. This is done in order to understand the position of object identification in the world of video processing.

2.2 Objects of interest

This brief section will explain which types of objects we will be interested in while doing the objects identification. Objects can be separated in two main categories: *static* objects and *moving* objects. For our purposes, static objects are those that have constant spatial and visual characteristics over time (e.g. books on a shelf). We will consider these to be part of the static background of the scene. These objects will not garner the focus of our attention since we are interested in the identification of *moving* objects. Moving objects are those that can change position and shape over time. As described by Brémond and Thonnat in [1], moving objects can be classified in three subcategories representing rigid and non-rigid objects and objects that cannot be defined by a model (we will call them freeform objects). The rigidity of objects is defined by the variation (or non-variation) of their shape. The shape of rigid objects in 3D space does not vary over time but non-rigid objects' shape does. This thesis does not differentiate between the different types of object rigidity since we are interested in identifying all types of moving objects.

2.3 Theory of video decomposition and reconstruction

In order to accomplish the identification of objects in digital video sequences, it is important to understand what video is composed of. This section therefore covers the decomposition and reconstruction of video. Note that the equations in this section have been derived to demonstrate that the moving objects identification algorithm developed in this thesis has a theoretical background.

The simplest way of defining digital video would be to say that it is a sequence of frames (a frame being an image), as shown by equation 2.1 below. In this equation, a and b can take different values depending on the continuity of the video. The theoretical limits of a and b would be $-\infty$ and $+\infty$ respectively, but for finite time video, a would be 0 and b would be greater than a. In both cases, n would represent discrete time, since frames are the representation of the visual aspects of the scene at a moment in time in the video.

$$Video = \sum_{n=a}^{b} Frame_n$$

Equation 2.1 – Basic video equation

The position and appearance of the collection of objects (moving or not) that appear in the frames define the similarities and differences that can be found by comparing individual frames. As explained in the previous section, these objects are

either static or moving. A frame can therefore be represented as a collection of static and moving objects as shown in equation 2.2. In this equation, n represents an arbitrary discrete time and $a(n)$ and $b(n)$ respectively represent the number of static and moving objects in the scene at discrete time n. $Static\ Object_{n,k}$ and $Moving\ Object_{n,l}$ respectively represent the k^{th} static object and the l^{th} moving object at discrete time n.

$$Frame_n = \sum_{k=0}^{a(n)} Static\ Object_{n,k} + \sum_{l=0}^{b(n)} Moving\ Object_{n,l}$$

Equation 2.2 – A frame made of static and moving objects

The previous section also mentioned that all static objects would be considered part of the static background image. A single entity representing the background at time n can then represent the collection of static objects at that same time as shown by equation 2.3.

$$Frame_n = \sum_{l=0}^{b(n)} Moving\ Object_{n,l} + Background_n$$

Equation 2.3 – A frame made of moving objects and a background

The $\sum_{l=0}^{b(n)} Moving\ Object_{n,l}$ is what *SmartID* tries to identify in every frame of processed video sequences. *SmartID* relies heavily on this equation and will therefore also need to deal with a background isolation process. *SmartID* will be covered in detail in chapter 3.

2.4 Overview of concepts related to object identification

In this section, we will provide an overview of concepts that are closely related to object identification, even if they may have different primary objectives than the ones that are addressed by object identification. Some of them may be used by object identification and others may make use of object identification.

2.4.1 Motion Detection

The whole purpose of video sequences (digital and analogue) is to communicate image differences over time. If there are no image differences between consecutive frames over time, then we have an image rather than a video sequence. The differences found in sequences of images can originate from different sources. When the video sequences represent such things as television channels, there are plenty of difference sources, including camera operations (panning, tilting and zooming), object motion, fading and scene changes. In this thesis, we are concerned with original video sequences that come from video cameras, and not with those that come from television sequences.

The main distinction between these two types of sequences is the presence of scene changes in the case of television sequences. As denoted by Hirzalla in [2], the presence of motion can easily be detected by comparing pairs of consecutive frames. He also explains that a basic comparison approach is to use simple parameters such as pixel intensity. When comparing two consecutive frames, if the difference between them

exceeds a certain threshold, then a scene cut may be identified. The difficulties surrounding the detection of scene cuts include the prevention of false cuts and missed cuts as Hirzalla demonstrates in [3] and [4]. He also describes in [5] that more involved cut detection such as the one performed by *SmartFrame*™ can be performed to detect different types of cuts such as gradual, sharp, flash or action. The *SmartFrame*™ technology can also be used to simply detect the presence of motion by applying its scene cut detection at a finer grain.

When leaving television sequences aside, we see that simple (non-edited) video camera sequences are driven mainly by just a few things: camera operations, object motion, camera noise and other visual events. Motion detection mainly focuses on globally detecting the motion of objects in a scene. In many cases, when the detection of motion is required, a fixed camera that does not support special operations is used, thus eliminating the need to deal with camera-generated motion. In these cases, the only visual disturbances that are left to filter out are the camera noise and possibly other visual events (e.g. changes in level of ambient light). In other cases, it may also be necessary to deal with camera motion but this will not be covered here.

There are many different techniques for detecting motion in video sequences, including the use of motion histograms ([6]) and the detection of boundary changes in regions of colours or luminance. By definition, the detection of motion needs to detect the temporal location of motion and not necessarily the spatial location. Hardware

motion detectors are a good example of that because they only provide two states at any given point in time: *motion* and *no motion*.

When the detection of motion is localized and assembled into objects, then it becomes object identification rather than motion detection. Most if not all algorithms that make use of moving objects identification techniques, directly or indirectly, also use some kind of motion detection algorithm.

2.4.2 Moving Objects Identification

The identification of moving objects consists of finding logical moving regions in a sequence of images. Depending on the algorithm used, these regions may be grouped to form what we call objects. Furthermore, other techniques might be used in conjunction with object identification to be able to screen the identified objects according to specific filtering parameters. Object identification by itself does not recognize the identified objects and does not associate newly identified objects with previously identified ones. These two functions are respectively done by object recognition and object tracking algorithms.

There is a wide range of ways to accomplish object identification and this section describes a few. Most techniques use concepts such as edge detection, frequency domain analysis and optical flow.

A popular method of finding moving objects is to use aggregation of optical flows. Optical flows are a representation of the instantaneous motion in a sequence of images as described in [7]. Every optical flow (motion field, or instantaneous motion) is applied to a section (from a single pixel to a group of pixels) of an image in a sequence in an attempt to determine the position of the section in the next image. All the optical flows in a single image applied to their associated sections in an image should result in the approximation of the next image. Optical flow algorithms are devoted to finding these motion fields. Many algorithms have been developed to calculate the optical flows and a number of these are presented by the authors in [8]. Once the optical flows have been calculated, then other aggregation techniques can be used to form objects from them. In [9] and [10], Yamamoto, Mae, Shirai and Miura identify moving objects by finding regions of similar optical flows within a specified window area. One drawback of using such a method is if the background motion is similar to the objects' motion. In such cases, the contour of the objects cannot be resolved by optical flows alone. Mae, Shirai, Miura and Kuno use a similar technique, but in addition, they combine edges to the optical flows in [11] to avoid the background movement problem.

Region segmentation is also a popular technique of finding objects in image sequences. It consists of determining groups of pixels in images that form logical regions based on a number of properties dependant on the algorithm used. In [12], Bascle, Bouthemy, Deriche and Meyer perform region segmentation by combining the benefits of a B-spline snake contour approach with those of motion-based region segmentation to

identify the moving objects in a video sequence. They use the motion-based region segmentation to initialize their snake-based contour algorithm. They then perform a motion-constrained matching of consecutive contour points in order to track the identified objects. Similarly, in [13] and [14], the authors find regions based on pixel colour and determine the object labelling by the position of the regions with respect to the identified edges.

More complicated approaches to object identification have also been explored such as the one presented by Létang, Bouthemy and Rebuffel in [15], in which they use a multiresolution wavelet transform on monodimensional time signals to determine the motion regions. These time signals come from the decomposition of an M by N pixels video sequence into M x N signals in time. When short period high frequency combined with longer period low frequency signals are detected, then they assume that there is motion at that coordinate in the M by N frame. A M by N motion map is generated from the integration of the processed M x N signals to determine the regions of motion. The greatest disadvantage of using wavelet-based algorithms is that they are very processor intensive.

The identification of objects in image sequences is generally used as a very important building block for other higher level image and video processing algorithms such as object recognition (described in section 2.4.5) and object tracking (described in section 2.4.6), regardless of the method used to find the objects. In chapter 3, the moving objects identification algorithm that is proposed is presented in detail.

2.4.3 Object Description

When an object needs to be described, we need a number of attributes that will accurately portray it. Of course, what the descriptions will be used for has a great impact on the attributes that are chosen to describe objects. Each attribute normally describes (but is not restricted to) either the temporal behaviour (e.g. velocity) or the spatial appearance (e.g. size) of an object. The set of attributes that is used to describe objects is what differentiates the object description techniques and depends on the type of algorithm or system the description technique will be combined with. Sections 3.4 and 5.4.4 discuss in more detail the attributes that are being used in the application developed in this project.

A number of object description models are used in the video processing field, of which edge models, corner models and vertex models are very popular. A combination of these types of models is often used in an attempt to better describe the objects. In [16] Blaszka and Deriche give a brief overview of an edge model, a corner model and a vertex model. Their edge model is based on the representation of a straight line with 5 parameters, taking into account the intensities of grey levels on both sides of the edge, the distance from the edge to the origin, a parameter representing the blur effect and the orientation of the edge. Their corner model is an extension to their edge model and describes the junction between two edges. The combination of parameters from two edges leads to a seven parameters description for a corner. Similarly, they describe a vertex (a triple junction) as the intersection between two adjacent corners, leading to nine

descriptive parameters. Obviously, these models alone are not sufficient to describe complicated objects, but by combining a number of these models together (i.e. a number of edges plus a number of corners), more complicated objects could be described.

In [17], Pope and Lowe define a learning image model. Given a set of learning images, a model that describes objects in the images is defined. Note that this paper only concentrates on the use of static images as opposed to video images. The descriptions are made of probability distributions describing the possible appearance of objects. After these descriptions have been derived, the model is later used for object recognition.

In [18], the authors discuss the use of multidimensional receptive field histograms in order to give a description of objects. A histogram is generated representing the probabilistic spectrum of colour occurrence. This proves to be a very effective way of describing objects if their shape can vary over time but their general appearance does not undergo considerable changes.

The description of objects is one of the supports for a number of higher level video processing algorithms such as object recognition and object tracking, but these rely on the matching of object descriptions, which is covered in the following section.

2.4.4 Object Matching

Object matching techniques are not used as complete solutions, but are rather used as building blocks for other image and video processing algorithms. Object

matching is a simple concept and can be described as follows. Let D be the description of an object through a set of attributes, and let $S(D)$ be a set of object descriptions. If one (or more) elements of $S(D)$ have the same or have similar attributes (matching attributes) as D, then an object matching algorithm should find it (them) in the set. If none of the object descriptions in the set matches the one from the object's description, then the algorithm should determine that there is no match. A confidence level is often associated with matches because of the generally noise tolerant results of the matches. This gives an idea of the reliability of the matching process.

The two main differences between matching algorithms are the attributes they use to describe the objects (see section 2.4.3 above) and the technique used to match the attributes. The matching techniques generally consist of finding the minimum distance between two object descriptions. This distance can be calculated in a number of ways and depends on the description attributes themselves.

In [18], Schiele and Crowley give a quick overview of histogram matching techniques for colour histogram based object description methods. These can easily be extended in a more general fashion to allow matching of parameters that are not related to colour histograms. One technique that is presented simply uses the sum of squared differences (commonly known as SSD) to determine the distance between two subsets of parameters. In order to extend the use of this technique to something that may not be related to colour histograms, normalization of individual parameters could be carried out before the SSD processing takes place. Schiele and Crowley define the SSD as shown in

equation 2.4, where H represents the current set of parameters to match, and T represents the set of parameters from a set of object descriptions.

$$SSD(H,T) = \sum_{i,j} (H(i,j) - T(i,j))^2$$

Equation 2.4 – Sum of squared differences

A second method they present refers to the χ^2 test, which can be calculated as shown in equation 2.5. It can be compared to a weighted SSD, with a weight inversely proportional to the attributes' value. This is likely to give better results than a regular SSD when the range of values the attributes can have is sparse within the same set.

$$\chi^2_{TH}(H,T) = \sum_{i,j} \frac{(H(i,j) - T(i,j))^2}{H(i,j) + T(i,j)}$$

Equation 2.5 – χ^2 test

Using one of these two techniques, a complete matching process would consist of calculating all combinations of H and Ts and then determining which combination results in the minimum distance. Obviously, distance thresholds are used to prevent the matching of unrelated objects (i.e. if the minimum distance found is too large, then the match is simply rejected).

Huttenlocher and Rucklidge have taken a different approach in [19], in which they compare images using the Hausdorff distance, which consists of finding the distance between a set of points *A* and a set of points *B*, where *A* is the set of points to match, and

B is a set of points from a model. That distance is defined by equation 2.6, in which two finite sets of points $A=\{a_1,...,a_p\}$ and $B=\{b_1,...,b_q\}$ are compared, and where $h(A,B)=\max_{a\in A}\min_{b\in B}\|a-b\|$.

$$H(A,B)=\max(h(A,B),h(B,A))$$

Equation 2.6 – Hausdorff distance

The authors further explain the significance of $h(A,B)$ as follows:

> *"The function $h(A,B)$ is called the* directed *Hausdorff distance from A to B. In effect, $h(A,B)$ ranks each point of A based on its distance to the nearest point of B, and then the largest ranked such point (the most mismatched point of A) specifies the value of the distance."*

They also discuss later in the paper that a partial distance approach can also be used to insert a certain degree of freedom in the calculations. This partial distance rejects the K largest values found in $h(A,B)$ and $h(B,A)$, often resulting in a smaller Hausdorff distance. $h(A,B)=\max_{a\in A}\min_{b\in B}\|a-b\|$ then becomes $h_K(B,A)=\overset{th}{K}_{b\in B}\min_{a\in A}\|a-b\|$.

In [20], Huttenlocher et al. improve the technique by introducing a translation process in which the model set of points can be translated in order to further minimize the distance between the sets as shown in equation 2.7.

$$M_T(A,B) = \min_t H(A, B \oplus t)$$

Equation 2.7 – Hausdorff distance improved by translation

Many more algorithms (Mahalanobis distance in [21], Softassign Procrustes Matching in [22], Modal Matching in [23], Informational distance in [24]) could be described in this section but this is out of the scope of this thesis. Some applications of object matching are discussed in the following two sections on object recognition and object tracking.

2.4.5 Object Recognition

In its simplest form, the recognition of objects implies the process of matching an object with one that is already known. The purpose of object recognition algorithms is to fulfill the matching process requirement and to define a description model for the objects that need to be recognized. It is important to make the distinction between perceiving an object and recognizing it. In [25], Edleman explains the difference between these as follows:

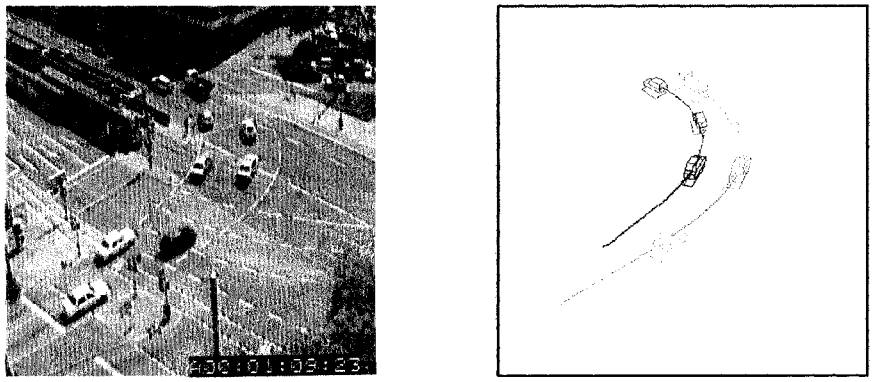
> *"Unlike "merely" perceiving a shape ... recognizing it as something involves memory, that is, representations of shapes seen in the past."*

The objects to be recognized usually originate from static images or from video sequences. One of the keys to object recognition is the generation of an accurate object

model that will properly describe the objects (or object types) to be recognized. Furthermore, an appropriate database containing the description of the objects that need to be recognized according to the object model needs to be defined.

Traditionally, object recognition was used on static images only. This implied that the notions of time and motion were not part of the object recognition equation. Nowadays, applications are also found in the video domain where moving objects are also recognized. This has led to improvements on the conventional visual models by using behavioural models as well. Many diverse types of object models have been defined in the literature since the models are what make the difference between object recognition techniques.

In [26] and [27], Koller et al. use a 3-D generic polyhedral vehicle model to recognize vehicles in a scene. They use 12 length parameters defining various aspects of different types of vehicles (e.g. bus, sedan and hatchback). From the edges that are extracted from the object (vehicle) to be recognized and a projection of the 3-D vehicle models on a 2-D plane, the type of vehicle can be determined through the use of the Mahalanobis distance. The authors noticed that the 2-D model and the object model were sometimes misaligned due to shadows, which led to the introduction of an illumination model that considers shadows. In addition to the recognition of vehicles, they also track the vehicles that they find. A recognition and tracking sample of their system taken from Koller in [28] is shown in figure 2.1 below.



(a) Input image

(b) Recognized cars with associated tracks

Figure 2.1 – Recognition and Tracking sample of model-based vehicle tracking

In [29] (with extensions in [30] and [31]), Sullivan et al. use a vehicle model similar to the one above to recognize vehicles in a scene. Their technique involves the understanding of the scene in three dimensions, including the camera location. Using such a priori knowledge makes it relatively easy for them to deal with classic recognition and tracking problems such as occlusions and interpretation of behaviour.

Gavrila and Davis are able to recognize human motion in [32] and [33]. With the help of multi-views (more than one camera), they can determine the location of objects contours and get a sense of depth, which let them use a generic 3-D human model that has 22 degrees of freedom.

In [34], Gavrila and Philomin propose 2-D pedestrian and traffic sign recognition that could be used in "*Smart*" vehicles. They use a (2-D) hierarchy model of the

pedestrians and traffic signs in order to recognize the objects in the scene. This change in model is explained by the fact that the multi-views approach above used multiple non-moving cameras and that they are now using a single camera in a moving car (i.e. equivalent to a moving camera).

In [35], Zaremba, Locas and Gougeon use different types of neural networks and image features to recognize tree species. This is a good example of recognition that does not rely on the motion of objects in order to recognize them. This work does not deal with video imagery but rather works on still high-resolution aerial images. In [36], an algorithm for the recognition of tree species is also presented in which high-resolution satellite images taken from different angles of view are analysed. The technique used in this report matches the combined visual information from a series of images taken at different view angles with those stored in a database.

While strict object recognition deals with the matching of objects to those in a usually predefined database, the matching of objects with those found in previous video frames could also be performed. This is the task of object tracking that is presented in the next section.

2.4.6 Object Tracking

Object tracking represents the spatio-temporal relationship of objects in video sequences and just like object recognition, is a direct derivative of object matching.

When the identification and description of the objects of interest have been performed, then the spatio-temporal object matching can be carried out.

Object tracking and recognition usually differ only in the set of objects that is used to match the objects that need to be tracked or recognized. Object recognition normally uses a predefined set of objects (or types of objects) to perform its recognition; whereas object tracking consists of finding newly identified objects in a set of previously identified ones (i.e. uses a dynamic set of previously identified objects).

The tracking of objects implies that time is part of the equation and therefore does not apply to static images. As for object recognition, the main task in defining an object tracking system is the generation of an accurate object model that will allow successful matching of the objects of interest. Although object tracking in a strict sense does not imply the recognition of the objects to be tracked, many systems in the literature combine the benefits of both concepts. For that reason, some of the work presented in this section relates to both object recognition and object tracking.

In [12], the authors rely on the fact that from frame to frame, the shape of tracked objects and the spatial location will not vary significantly. As was explained in 2.4.2, they use snake-based contour tracking and region-based motion analysis ideas to perform the tracking. The region-based motion analysis estimates the location of the object in the following frame, and then in an optimistic fashion, the snake-based contour is used to determine the new shape of the object.

In [37], Smith first identifies the moving objects (vehicles) by using segmentation of optical flows. The resulting clusters are then described using six parameters including the bounding box, centroid and motion model. The clusters are then matched to a list of previously seen clusters, allowing tracking to be achieved.

In [38], Irani, Rousso and Peleg identify the moving objects by isolating a single motion from an image pair and iteratively do so with the remaining motions until all objects have been found. The tracking of the objects is performed through a motion cancellation step that basically aligns the object with its previous representation. Each found object is then refined (sharpened) by performing a weighted average of previous motion-compensated images, which has the effect of blurring everything but the object of interest.

In [39], Bell, Felzenszwalb and Huttenlocher perform moving objects detection through the use of an image motion registration process. The use of such a technique lets them compensate for camera motion while facilitating the discovery of moving objects in a video sequence. The identification of the objects is done through the segmentation of the images based on colours and also based on the motion magnitude. Once the segmentation has been completed, two separate object trackers are used, for short-term and long-term tracking. First, the short-term tracker is in charge of finding potential moving objects. It does this by matching the size and location of new or potential moving objects with those that have been seen in past frames. When a candidate object is detected in a certain number of frames, then it is passed to the long-term tracker. The

long-term tracker uses a generalized Hausdorff distance measure from [20] for the matching based on the composite motion and edge information of the objects. The object model is improved from frame to frame to make the tracking process more robust.

2.5 Summary

In this chapter, we have made an outline of the theory and applications surrounding the object identification concept. We have first given a brief overview of video decomposition in order to understand the theory that is leveraged in the algorithm developed in this thesis. We have also determined that all categories of moving objects (rigid, non-rigid and freeform) will be of interest to us. A number of concepts related to object identification have also been discussed.

Most concepts in this chapter are tightly related since they potentially all deal with moving objects. Motion detection discovers their presence, object identification determines where they are located, object description generates a model for them, object matching allows us to find similarities between them, object recognition tells us that it recognizes them, and object tracking follows them in time and space. Furthermore, higher level algorithms (matching, recognition, tracking) are almost always dependant on lower level ones (motion detection, object description, object identification), as is shown in figure 2.2 which depicts an inclusive dependency graph. This makes it worthwhile

developing generic algorithms at a lower level to broaden the range of applications for them.

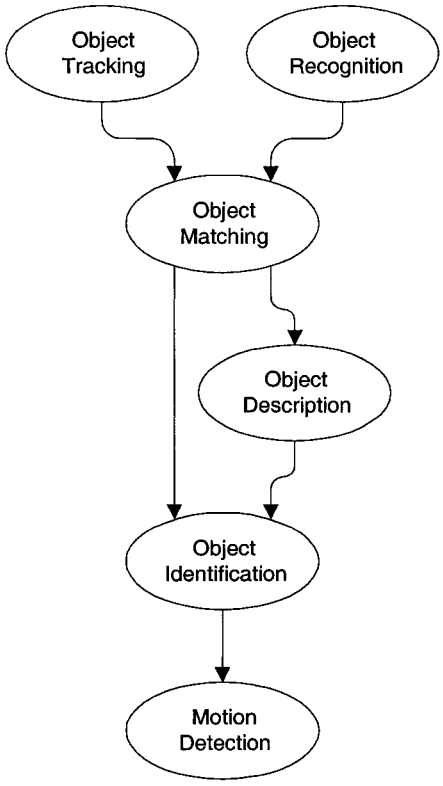


Figure 2.2 – Inclusive dependency graph of moving objects related algorithms

Chapter 3

Development of the Moving Objects Identification Algorithm

3.1 Introduction

The main goal of *SmartID* is to focus on identifying arbitrary moving objects in digital video sequences. *SmartID*'s basic concept is fairly simple and is separated in two steps:

1- Determine the most up to date static background from incoming video frames,

2- Find moving objects in newly arrived frames by comparing those with the previously determined static background.

This concept simply takes advantage of the most basic definition of motion, which is the process of changing position. Applied to video, this definition says that any change in a sequence of frames over time represents something that has changed position (i.e. that is not static).

A mask of areas that have changed in the newest video frame can be derived. The application of that mask over the latest frame would result in the visual representation of the changed areas in the image. Since *SmartID* works at the object level rather than with areas of motion, the output of the algorithm is therefore a mask of moving objects in the current frame accompanied by a bitmap of the most up to date background. As it is the case for areas of motion, objects can later on be recovered by masking the current frame with the objects mask. According to equation 2.3, the whole original frame can be reconstructed by overlaying the extracted objects on top of the background bitmap.

Figure 3.1 shows a simplified high level architecture of *SmartID*. In this figure, F_n represents an incoming video frame at discrete time n, M_n represents the mask of moving objects also at discrete time n and BG_n and BG_{n-1} represent the isolated background image at discrete times n and $n-1$ respectively. From this figure, we can see that the process of isolating the background uses feedback (i.e. using the previously derived background, BG_{n-1}, to assist in finding the newly derived one, BG_n). Note also

that the search for moving objects process also requires a previously defined background (BG_{n-1}). There is an implication from this diagram that previously derived backgrounds are always well-defined, even at the starting time boundary, thus showing *SmartID* in a steady state.

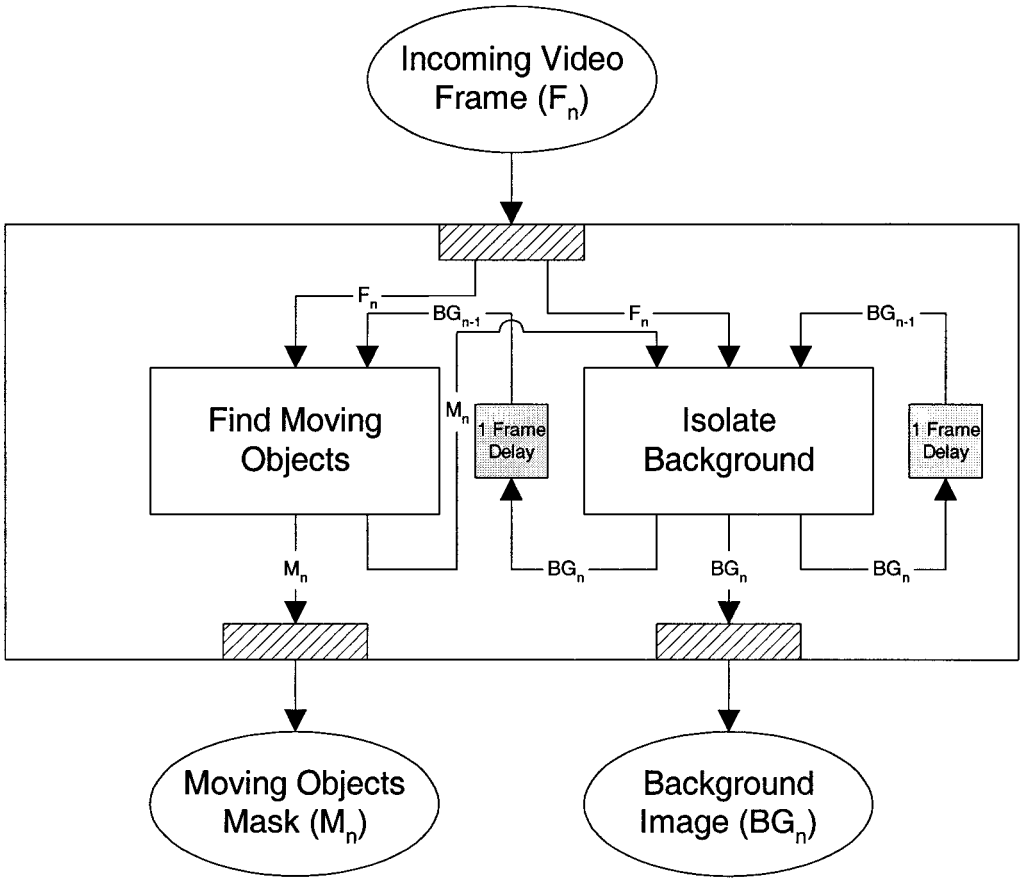


***Figure 3.1 – Simplified high level architecture of* SmartID**

Also from figure 3.1, we can see that once in the steady state, the moving object search has to be performed before the background isolation since the latter uses the result of the former to lead to its own result. This does not contradict the order of the previously described *SmartID* steps since before reaching the steady state, *SmartID* first

has to define a background bitmap to start with. The first background can be defined in a number of ways, but the ones treated here are as follows:

1- The background can start as a black image (or any other arbitrary colour) and eventually adapt to the incoming video images. This has the disadvantage of often creating false movements the size of the image at start-up time since the difference between the first frames and the black background are normally quite large.

2- It could also start as a duplicate of the first video frame received by *SmartID*. The advantage of this method is that the algorithm reaches its steady state quite rapidly when there are no real moving objects in the first frame. When this is not the case, then moving objects will only be detected in subsequent frames and the background will have "object shadows" that will disappear over time as the background adapts to the incoming video frames.

The details on how moving objects are identified and how backgrounds are generated are presented and explained in the following sections.

3.2 Identification of moving objects

As stated in the previous section, *SmartID* relies on the assumption that it always has a valid background image available. Keeping that assumption in mind and according

to equation 2.3, the process of identifying moving objects becomes quite straightforward. The inputs and output of the moving objects identification process are represented in figure 3.2.

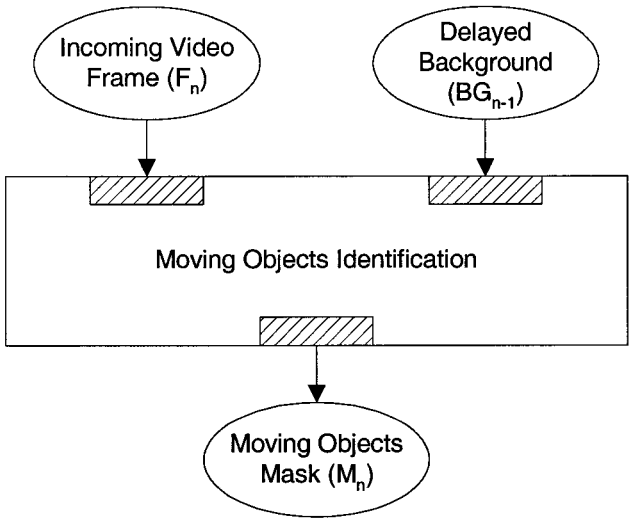


Figure 3.2 – Inputs and output of* SmartID*'s moving objects identification process

SmartID's inputs at discrete time n are the incoming video frame, F_n, and the delayed background, BG_{n-1}, while its output is the moving objects mask, M_n. The incoming video frame is the newest frame that is available from the sequence, while the delayed background is the image of the background that was isolated during the processing of the previous incoming frame (F_{n-1}). Section 3.3 explains the background isolation process. The moving objects mask is a pixel block mask (with arbitrary block size) representing regions where the identified objects were found. Every object region is assigned a different number in the mask in order for objects to be separately identified.

An example of such a mask is given in figure 3.3 where three separate objects are assigned separate mask values and where the static portion of the image is assigned zeros.

Figure 3.3 – Moving objects mask

The first step in identifying moving objects is to find pixels in the incoming frame that are different from the background. According to theory, anything that is static will not visually vary over time. We can extrapolate this and make the reasonable assumption that anything that visually varies over time is not static (i.e. is moving). This is only stated as a reasonable assumption since visual differences could also be attributed to general environmental changes not related to object motion. This relationship is stated in equation 3.1, which reformulates equation 2.3. In this equation, n represents a discrete time, and the pair n,l used in $Moving\ Object_{n,l}$ represents object l at discrete time n.

$$\sum_{l=0}^{b(n)} Moving\ Object_{n,l} = Frame_n - Background_n$$

Equation 3.1 – Moving objects

Although this theoretical equation uses a background from time n, the practical solution used by *SmartID* only has access to a background at time $n - 1$. This does not pose a problem as long as the delta between $Background_n$ and $Background_{n-1}$ is negligible. Since the delta between these two is usually attributed to noise and that *SmartID* filters out that noise in the difference frame, then equation 3.1 could be rewritten using $Background_{n-1}$ instead of $Background_n$ with minimal effect on the objects found. The equation would then become an approximation of $\sum_{l=0}^{b(n)} Moving\ Object_{n,l}$ as shown in equation 3.2.

$$\sum_{l=0}^{b(n)} Moving\ Object_{n,l} \cong Frame_n - Background_{n-1}$$

Equation 3.2 – Moving objects approximation

The first step in identifying moving objects is therefore done by a pixel-wise absolute difference between the incoming frame and the delayed background image. This serves the purpose of finding pixels that are "in motion". A theoretical visual example of this (using $Background_n$) is depicted in figure 3.4.

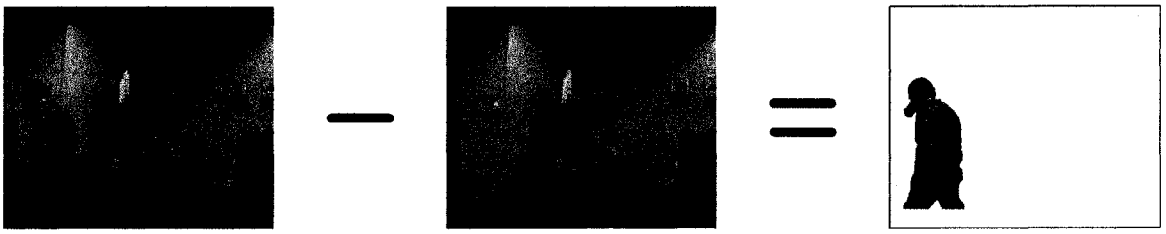


Figure 3.4 – Extraction of objects from original and background images

As mentioned above, this works well in theory, but in practice, there will be injected noise from the use of the background at discrete time *n-1* instead of the one at discrete time *n*. Many factors may be responsible for these differences and all of these add up to the overall noise level in the pixel-wise absolute difference frame. Here are a few examples of such noise sources:

- Camera quality: The quality of the video camera makes a great difference in the resulting digital video quality. If we looked close enough, we could visually observe differences in consecutive digital video frames (before digital compression and decompression loss) even if there were no apparent differences in the original scene. Some cameras are better at preventing this kind of noise than others.

- Illumination effects: This kind of noise is generated by real, sometimes very small changes in the illumination of moving and static objects in the scene. This type of noise can easily be seen when capturing video in an environment with fluorescent lights.

- Digital compression loss: When working with video that has been previously encoded and decoded by a lossy video CODEC (coder-decoder), then more differences may appear between consecutive frames. As an example, in H.263 video, interframes are encoded based on previous interframes and key frames to give an approximation of their respective original frame. This source of noise could be avoided if the video processing would be performed at the source of digital video (before any type of lossy encoding and decoding).

Figure 3.5 shows a visual example of unwanted noise. The pictures on the left represent two consecutive frames of a video sequence that do not contain any type of motion. The image on the right represents a mask of pixels that differ between the two original frames. These differences are represented by grey level pixels on the image for which darker levels represent greater absolute pixel differences.

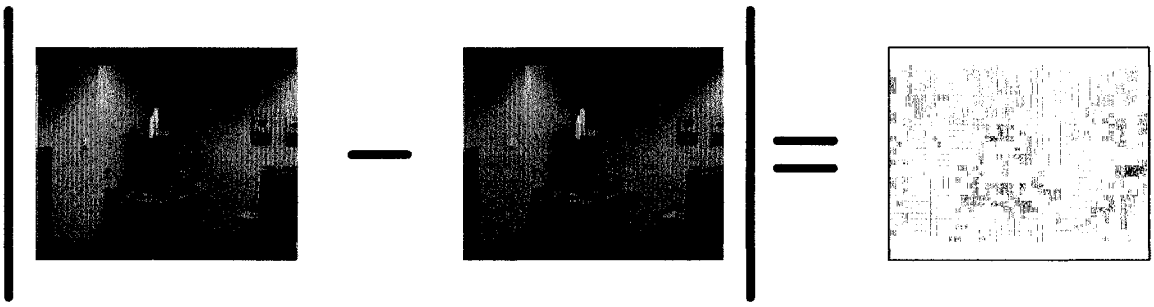


Figure 3.5 – Unwanted noise

Since this noise is unwanted, then the second step is to filter it out. 2D digital noise filters already exist for this kind of problem, but they generally require a fair amount of processing power. Because of the real-time constraint that is imposed on *SmartID*, these 2D noise filters are not used here. As a replacement, a series of customized filters have been designed and are discussed in the following pages. These filters are such that the processing power requirements can be adjusted easily when implemented.

A block filtering process accomplishes the first noise reduction step. *SmartID* separates the absolute difference frame in blocks of an arbitrary size (user defined, but typically 8x8 pixels). It then makes two observations on every one of these blocks. The first is the count of pixels that have non-zero absolute difference (*CNZADB*). The second is the sum of absolute differences in the block (*SADB*). It then compares these observations against arbitrary (user defined) thresholds. If both thresholds are exceeded (i.e. *CNZADB* > *CNZADB threshold* and *SADB* > *SADB threshold*) then all the differences in that block are kept intact, but if at least one of these criteria is not met, then all difference pixels within that block are reset to zero. The non-zero count threshold is defined as a percentage so that it is block size independent.

Internally, this percentage is then converted to a real value that depends on the block size (i.e. *CNZADB threshold* = *block width* × *block height* × *CNZADB threshold percentage*). Similarly, the sum of differences threshold is usually defined as an average absolute difference so that it is kept independent of block size. It is

then converted internally to a sum of differences threshold (i.e. *SADB threshold* = *block width* × *block height* × *average difference*).

Small absolute differences are normally attributed to one of the noise sources since they show up when newly arrived pixels are visually very close to background pixels. The sum of differences threshold condition takes care of getting rid of these noise sources. On the other hand, a block could have just a few pixels with high absolute difference and it would still meet the sum of differences requirement. We want to get rid of these blocks since they usually represent noise pixels that are scattered / sporadic, implying that they are unlikely to be part of moving objects. These types of blocks are filtered out by the non-zero absolute difference count condition since, on average, most pixels are not directly affected by noise. Algorithm 3.1 shows a high level algorithm of this block wise noise filter written in pseudo-code.

Multiple levels (typically 2) of this filter can be applied successively to the same absolute difference matrix with different block sizes. It makes sense to cascade these filters in such a way that the first one has the largest block size and the last one the smallest block size. Larger block filters represent coarser grain filters whereas smaller block filters represent finer grain filters. Cascading has the effect of first coarsely defining regions of interest (where differences are more significant) and then to more precisely define the edges of these regions and also removing more unwanted noise.

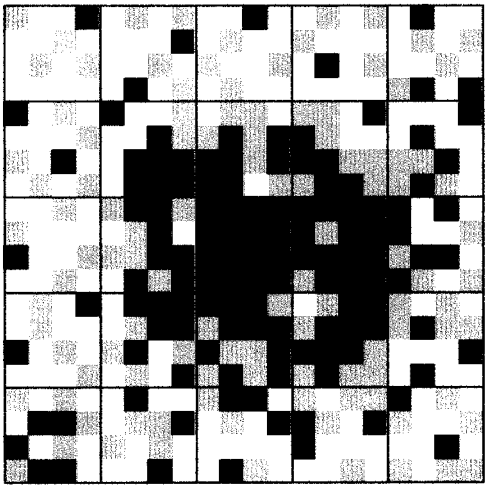
```
CNZADBTh = Count of Non-Zero Absolute Differences per Block Threshold
SADBTh = Sum of Absolute Difference per Block Threshold
CNZADB = Count of Non-Zero Absolute Differences per Block
SADB = Sum of Absolute Differences per Block
M = Matrix of absolute differences
bh = Block Height
bw = Block Width

for all the different blocks of size bh x bw in M
{
        SADB = 0
        CNZADB = 0
        for all values in block
        {
                if (value > 0)
                {
                        SADB = SADB + value
                        CNZADB = CNZADB + 1
                }
        }
        if ((SADB < SADBTh) or (CNZADB < CNZADBTh))
                zero all values in block
}
```

Algorithm 3.1 – High level algorithm of this block wise noise filter

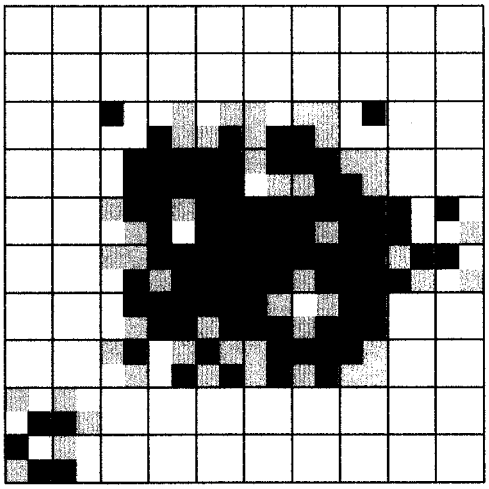
Figure 3.6 shows an example of the effect of cascading two filters on a 20 x 20 frame of absolute differences. For demonstration purposes, this figure only contains three levels of intensity (0 = white, 1 = grey and 2 = black).

Now that the block wise noise filters have been applied, most of the noise should have been cleaned. This means that if our last filter had small enough block size, then we could now make a mask of blocks that passed through the filter and work from there to pursue the object identification process. This mask is made of ones and zeros, where a

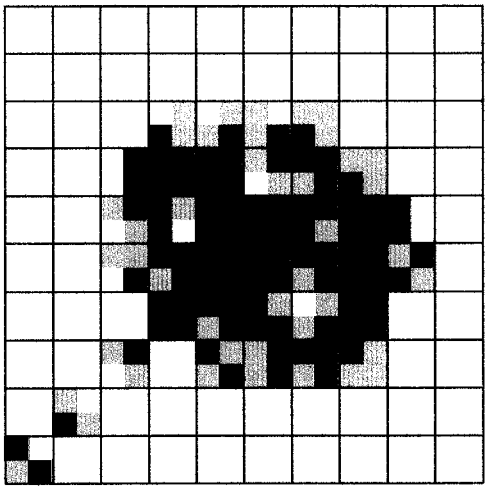
one represents a block that passed through the filters and a zero represents one that did not. Figure 3.7 shows the relation between the block mask and the difference frame.



(a) Original frame of absolute differences with overlayed 4x4 block layout for first pass filtering (SADBTh = 16, CNZADBTh = 8)

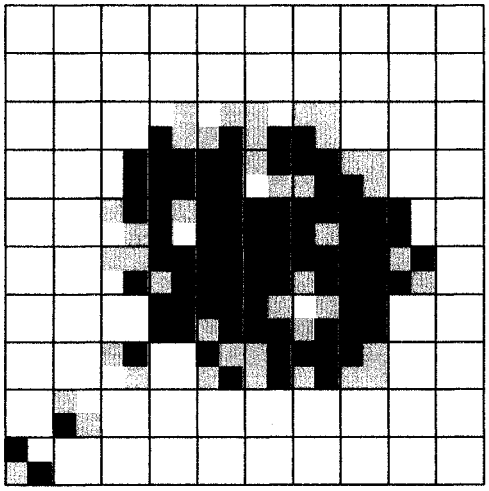


(b) Resulting frame of first pass filter from (a) with overlayed 2x2 block layout for second pass filtering (SADBTh = 4, CNZADBTh = 2)



(c) Resulting frame of second pass filter from (b) with the overlayed 2x2 block layout that was used

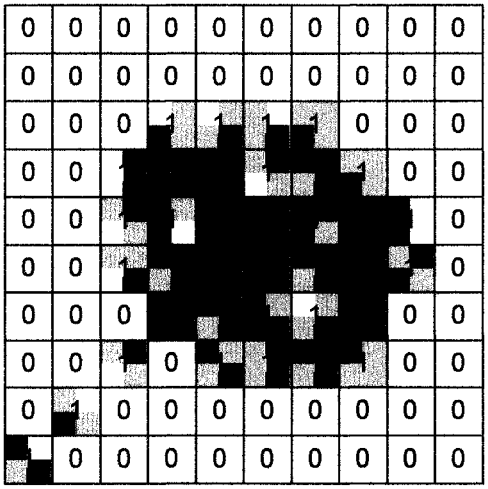
Figure 3.6 – Cascade of block wise noise filtering



(a) Resulting frame of second pass filter from figure 3.6(b) with the overlayed 2x2 block layout that was used

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	0	0	0
0	0	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0
0	0	0	1	1	1	1	1	0	0
0	0	1	0	1	1	1	1	0	0
0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0

(b) Mask of blocks that succesfully went through the two cascaded filters of figure 3.6



(c) Mask of blocks that succesfully went through the two cascaded filters of figure 3.6 overlayed on the filtered frame

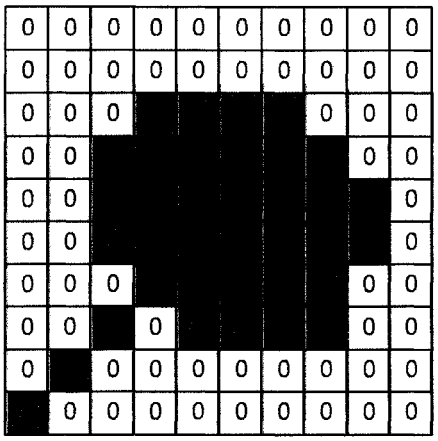
Figure 3.7 – Generation of block mask from filtered frame

Even at this point, some noise may not have been filtered, typically originating from scattered blocks of noise, edge effects wide enough to pass the block filters (edge effects may appear on abrupt changes in colour along physical boundaries), or even

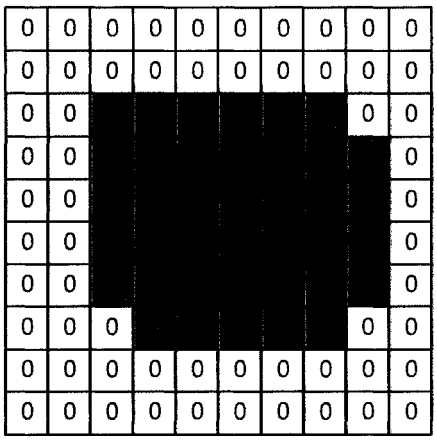
objects shadows. In order to attempt the removal of some of this noise and at the same time smooth out object edges, the next step of the algorithm clears and restores the mask of blocks.

The first part of this process is the cleaning. It simply clears all boundary blocks from the mask, that is, all blocks that have at least one "empty" block as their neighbour are also emptied (nulled). Diagonal neighbours may or may not be considered as neighbours and should be left as a user defined parameter. The effect of this cleaning process is twofold: first, it clears all blocks that are not part of a group of blocks and also blocks that are part of "artefact" groups of blocks (e.g. a line of blocks is unlikely to be part of an object). Secondly, it smoothes out the edges of groups of blocks, removing noise around them.

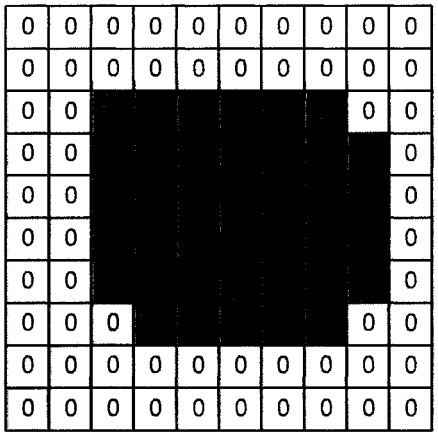
Since the cleaning process may have removed good boundary blocks from groups of blocks, then we must attempt to restore them as well as possible. This is done by the expansion of groups of blocks. Every block that has a number of empty neighbours must make these neighbours non-empty ones. This also has the effect of smoothing the edges of groups of blocks. As it was the case for block filters, the cleaning and restoring processes can be applied a number of times in series on the same mask, as long as all expansions are made after all cleanings. Furthermore, we could expand more than we clean to make sure that we don't lose any boundary details from the groups of blocks (typical: clean once, expand twice). Figure 3.8 demonstrates an example of the process of cleaning and expanding a mask of blocks.



(a) Mask of blocks from figure 3.7(c) where blue blocks will be cleared through the cleaning process and where green blocks will remain



(b) Mask of blocks representing the result of the cleaning process from (a) in green and blocks that will be added from the expansion process in blue

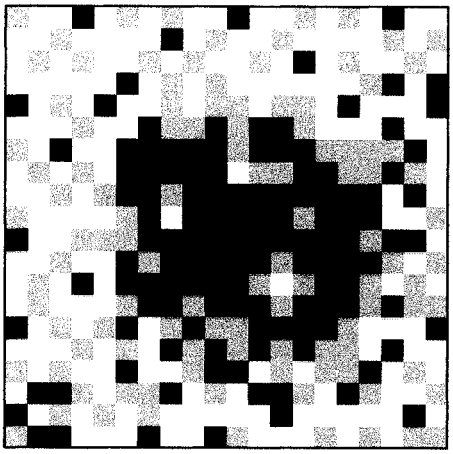


(c) Mask of blocks resulting from the expansion process in (b) with green blocks representing intact blocks from original mask, blue blocks representing new ones and yellow blocks representing cleared ones

Figure 3.8 – Example of cleaning and expanding a mask of blocks

This whole cleaning and restoring process has a few effects on the identification of objects. By cleaning (removing) the boundary blocks, we remove noise that may show up around the objects, but we also may end up removing blocks that are part of the

objects. That is usually a small price to pay since the restoration process restores most of these removed object blocks. Figure 3.9 shows the result of the whole filtering process (2 cascaded filters, cleaning and expanding) that started in figure 3.6.



(a) Original frame of absolute differences

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0
0	0	0	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

(b) Result of filtering (a) twice, making a 2x2 block mask of it and then cleaning and expanding the mask

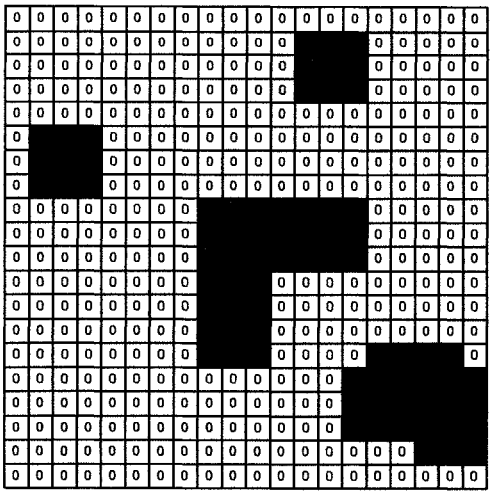
Figure 3.9 – Result of the whole filtering process, which started in figure 3.6

Again, the restoration process is not without disadvantages. By enlarging (restoring) the object blocks, some boundary definition (precision) is lost. This is done to ensure that most object blocks are part of the object mask. It is better to have a bit more than the objects themselves than to miss parts of them.

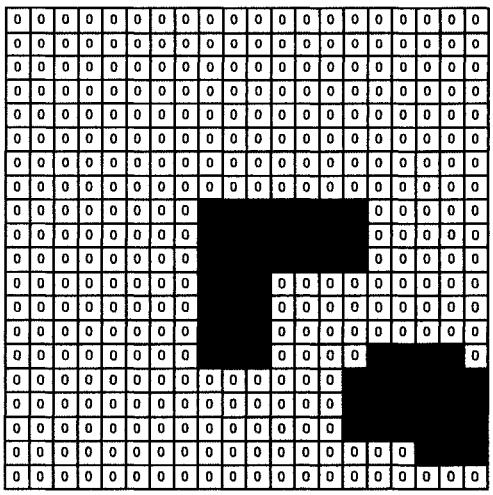
At this point in time, the mask should reflect the "space occupation" of the objects in the original frame. What is left to do is to filter out the objects that are too small to be real objects (they may simply be an agglomeration of noise blocks), and then to tag the resulting objects in the mask. To filter out small objects in the mask, we simply have to

count the number of blocks that are part of an object, and if that number is too small, then we simply discard the object by removing its blocks from the mask. By definition, if two or more blocks are neighbours, then they belong to the same object, and an object size is determined by its total count of neighbouring blocks. The tagging process uses the same neighbouring definition to assign numbers to the objects. All blocks that are neighbours have the same assigned number. Furthermore, all groups of neighbouring blocks must have unique numbers assigned to them (i.e. no two groups of blocks can have the same number). Figure 3.10 shows an example of a size filter followed by the object tagging process.

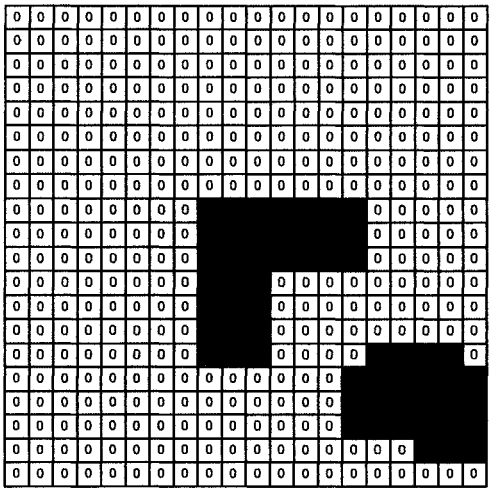
The mask resulting from this whole filtering process represents the mask of objects in the current frame, and is one of the two outputs of the algorithm. To get a visual representation of the objects found, we would simply have to apply the objects mask to the original frame. The main advantage of such a process is that no a priori information about the objects is required (i.e. no object type database). This is quite practical since we are not trying to recognize the objects in the video stream, we simply want to identify where the objects are located. A conceptual flow chart of the complete object identification process is shown in figure 3.11. The following section now discusses in detail how the background that was originally used by the object isolation process is determined.



(a) Mask of blocks that contains 4 objects



(b) Mask resulting from size filtering (a) with minimum object size of 10 blocks



(c) Mask resulting from numbering the objects in (b)

Figure 3.10 – Size filtering and tagging of masked objects

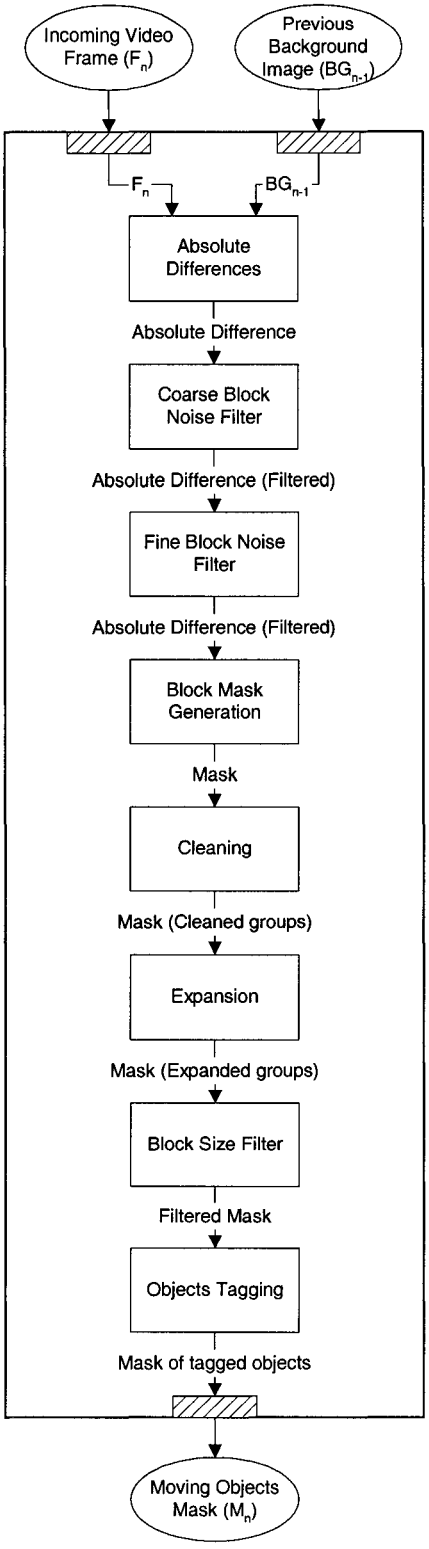


Figure 3.11 – Object identification conceptual flow chart

3.3 Background isolation

The background isolation process tries to keep an image of what the background in a video sequence looks like at any given point in time. Figure 3.12 shows the inputs and output of the background isolation process represented as a black box, and this section explains what it contains.

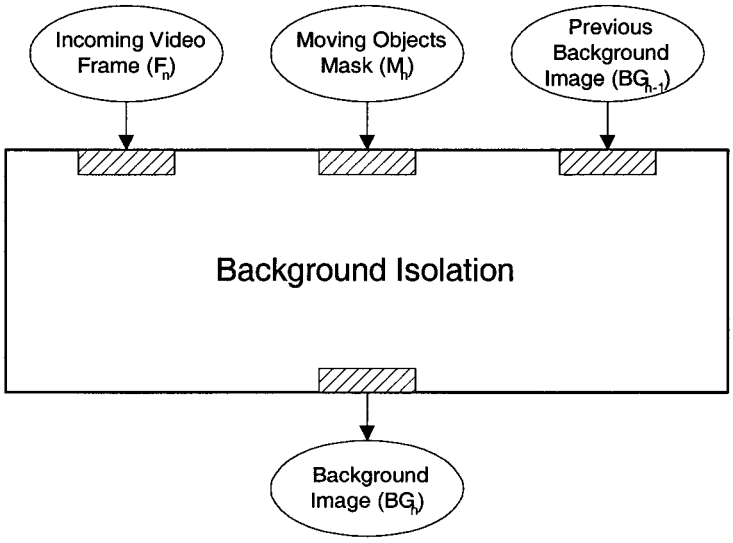


Figure 3.12 – High level background isolation process

The basic idea behind background isolation comes from reorganizing equation 2.3, which states that a frame is composed of its objects and its background. This is shown in equation 3.3, which says that a background can be derived by removing the objects from a frame.

$$Background_n = Frame_n - \sum_{l=0}^{b(n)} Moving\ Object_{n,l}$$

Equation 3.3 – Isolating the background

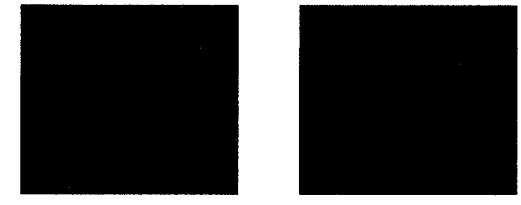
We can immediately see two potential problems arising from this equation when applying it literally:

- The background would be missing regions where objects are located.
- Moving objects that have become static can never be part of the background and will therefore always be misinterpreted as moving objects.

This implies that it must be modified to address these problems. The first modification that we can apply is one that would fill the missing regions in the background at time *n* that are covered by the objects at time *n*. Since it is impossible, from the frame with objects, to determine the background that is located behind the objects, then we can consider using the latest background information we had for these missing regions. This means that background *n-1* will fill the gaps. This is denoted in equation 3.4, which becomes an approximation rather than an equality. This principle is also demonstrated by figure 3.13.

$$Background_n \cong Frame_n - \sum_{l=0}^{b(n)} Moving\ Object_{n,l} + Partial\ Background_{n-1}$$

Equation 3.4 – Filling background holes with a previous background



(a) Image of background $_{n-1}$ (b) Image of Frame$_n$ with two objects in it (shown in red)

(c) Image of Background$_n$ where the green part has been taken from (b) and where the objects from (b) are replaced by the background from (a)

Figure 3.13 – Filling background holes with a previous background

To address the problem of new static objects, *SmartID* keeps a history of pixels that are not equal to the background. When it finds pixels that have been stable for a certain number of frames, then they can be integrated in the background. We call this process a static background update. A pixel is defined as being stable if its value does not deviate by more than a specified threshold over a number of frames. When this requirement is meet, then the background can be updated with it. When the stability threshold is not respected, then the stability timer is reset.

Figure 3.14 shows an example of a sequence of frames containing objects. One of the objects is moving, another one is changing appearance and the other one is not moving. In this figure, the state of the stability timer is shown directly in the objects. Suppose in this example that the stability threshold value is 5. At discrete time $n+5$, the

background would be updated with the yellow object from $Frame_{n+5}$, which has reached a stability value of 5.

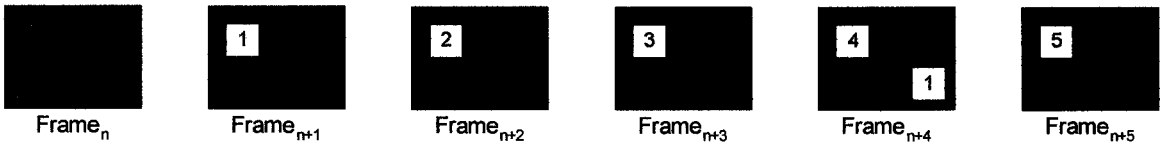
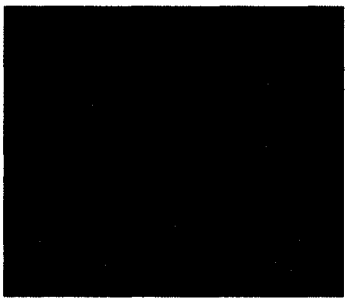


Figure 3.14 – Sequence of frames containing different types of objects

With this static update added in the process, the estimated background becomes a fairly good representation of the real one, but the process can still be improved by adding a background noise reduction method to it. We can easily imagine that the noise found in backgrounds would in theory fade away if we took a pixel-wise average of a number of backgrounds. On the other hand, we would not want small changes to the background (not generated by objects) to be lost by the averaging over time. *SmartID* improves the generation process by blending the new background with the previous ones. This has the effect of minimizing noise over time because of the averaging and also has the effect of adapting to small variations in the background (e.g. gradual fading of ambient light). This implies that not only previous backgrounds are used to fill holes created by objects in the newest background, but they are also blended with the newest incoming frames (with objects cut out) for a noise reduction effect. Furthermore, *SmartID* applies the same noise reduction principle to static background updates by blending newer differing pixels with older ones. This background noise reduction mechanism is shown in figure 3.15, which is a version of figure 3.13 that has been improved by noise reduction.

This figure does not show the noise reduction that would be performed on the static background updates, but the same type of blend would be performed. The blending process of an arbitrary pixel is demonstrated by equation 3.5, in which the *New Pixel Value* at time *n* represents the value of the pixel in $Frame_n$ and where *V* (at discrete times *n* and *n-1*) represents the blended average of the value of the pixel.



(a) Image of $background_{n-1}$

(b) Image of $Frame_n$ with two objects in it (shown in red)

(c) Image of $Background_n$ for which the most part is a blend of the green from (b) and the blue from (a) and where the objects from (b) are replaced by the background from (a)

Figure 3.15 – Background noise reduction through blending

$$V_n = New\ Pixel\ Value_n * BlendRatio + V_{n-1} * (1 - BlendRatio)$$

Equation 3.5 – Averaging through blending

A conceptual flow chart of the complete background estimation process is shown in figure 3.16.

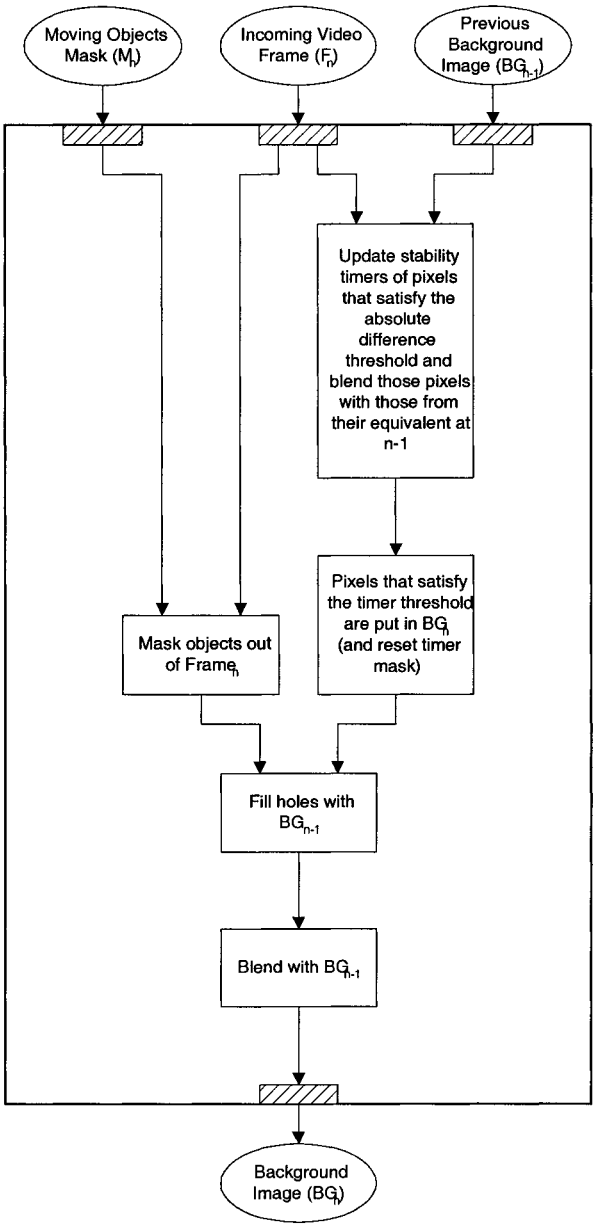


Figure 3.16 – Background isolation conceptual flow chart

3.4 Supporting description of objects

As it has been briefly discussed in chapter 2, the description of objects is usually the base of object matching and consequently of object tracking and object recognition as well. In the general case, the objects can be in motion or can be static, but for our needs, attributes for moving objects will be covered. Note that the parameters that will be described in this section represent only one possible set of parameters. We think that this set of parameters would apply well to an object tracking or an object recognition system that would use *SmartID*. This section was included to give the reader a general understanding of what the description of objects could be and is not intended as an object description solution that suits all purposes.

As discussed in section 3.2, the visual information about the objects of interest at discrete time n can be extracted by masking the input frame (also at time n) to *SmartID* with the moving objects mask resulting from *SmartID*'s processing.

What follows is a description of what the parameters that we cover here represent with respect to the objects to be described.

Bounding Rectangle, Height and Width: The bounding rectangle of an object is defined by the extents of the object with respect to the x- and y-axis. These extents are the minimum and maximum x and minimum and maximum y positions of pixels found in the object. This bounding rectangle can be used to calculate the height and the width

of the object as well as the middle point (centre) of the bounding rectangle. These different values give us an idea of the position of the object as well as a general perception of its size. Figure 3.17 shows an example of these parameters graphically.

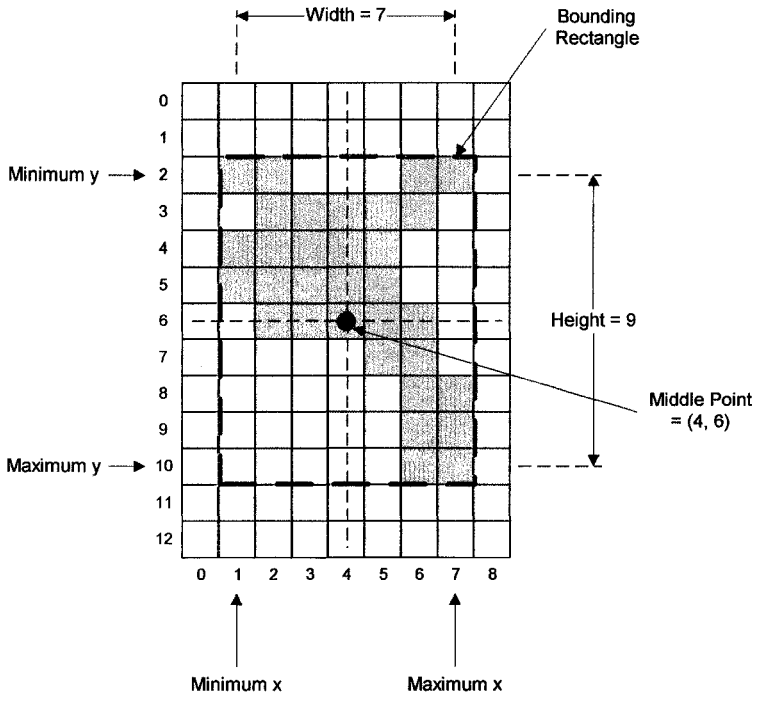


Figure 3.17 – Graphical representation of bounding rectangle parameters

Area: The physical area of the object is simply defined by the number of pixels it contains and is smaller than or equal to the result of multiplying the height by the width of its bounding rectangle. This gives a better idea of the size of the object. As an example, the area of the object in figure 3.17 is 32 units whereas the bounding rectangle's area is almost twice the size at 63 units.

Centre of Area (Centroid), Major Axis Length and Minor Axis Length, Rotation and Density: A few object parameters can be derived from the parameters of the *approximating ellipse*. These represent an ideal ellipse with moments of uniform area that are identical to those of the object (considering it has a uniform density of 1), and are very useful in giving a general idea of the position, orientation, size and aspect ratio of the object. Considering that the weight of every pixel in an object would be uniform, the centroid would represent the centre of mass of the object. This gives an even better idea of the position than the bounding rectangle did since it only takes into account pixels that are part of the object. Furthermore, the approximating ellipse and the object have the same centroid. The major axis length and minor axis length represent respectively the longest and shortest diameters of the ellipse. The rotation represents the angle of the major axis of the ellipse with respect to the horizontal axis, represented as an acute angle. The density is the unit of mass per unit area as defined by Stewart in [40] and is uniform in the case of the ellipse. Figure 3.18 shows these properties graphically where θ is the rotation angle of the major axis of the ellipse with respect to the x-axis.

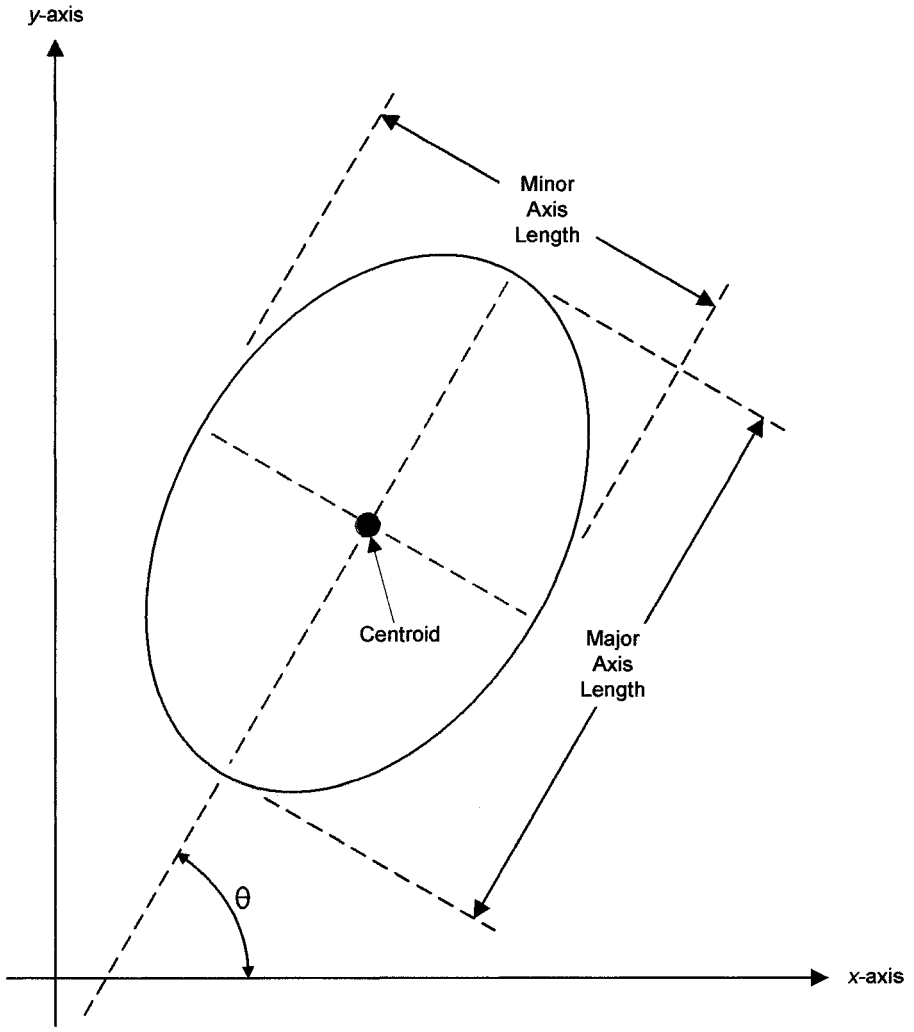


Figure 3.18 – Graphical representation of parameters of an ellipse

Colour Mean and Colour Variance: The previous parameters gave us information on the position, orientation, size and shape of the objects to be described, but none of them took into consideration the colour aspect of the objects. This could be very useful in distinguishing between two objects of similar shape, size and orientation. For example, suppose you have two differently coloured balloons of same size that you want to describe. Without considering the colour aspect of these balloons, they

would be described in exactly the same way using the above parameters since they have the same shape and size. When taking into account the colour of the objects, they can then be differentiated. The mean colour simply represents the average colour of the object while its colour variance gives us an idea on the colour spectrum diversity.

Now that these parameters have been defined, we need to know how to calculate their values. McKerrow in [41] gives us the formulas for the parameters of the ellipse (Centroid, major and minor axis, rotation and density), and other formulas are obvious. We have simplified McKerrow's formulas by making the assumption that the scaling factors in vertical and horizontal orientations are 1. These formulas are enumerated in equations 3.6.

$$Area = \sum_{\forall pixels} 1$$

$$Colour\ Mean = \frac{\sum_{\forall pixels} v}{Area}, \text{ where } v \text{ is the colour value}$$

$$Colour\ Variance = \frac{Area \times \sum_{\forall pixels} v^2 - \left(\sum_{\forall pixels} v\right)^2}{Area^2}, \text{ where } v \text{ is the colour value}$$

$$Centroid\ x = \frac{\sum_{\forall pixels} x}{Area} \qquad Centroid\ y = \frac{\sum_{\forall pixels} y}{Area}$$

$$Major\ Axis = \sqrt{\frac{a+b+e}{2f}} \qquad Minor\ Axis = \sqrt{\frac{a+b-e}{2f}}$$

$$Rotation = \frac{1}{2}\tan 2(2c, a-b) \qquad Density = \frac{4}{\pi}\frac{Area}{2f}$$

where

$$a = \frac{4}{\pi}\left(\sum_{\forall pixels} x^2 - \frac{\left(\sum_{\forall pixels} x\right)^2}{Area}\right) \qquad b = \frac{4}{\pi}\left(\sum_{\forall pixels} y^2 - \frac{\left(\sum_{\forall pixels} y\right)^2}{Area}\right)$$

$$c = \frac{4}{\pi}\left(\sum_{\forall pixels} xy - \frac{\sum_{\forall pixels} x \sum_{\forall pixels} y}{Area}\right) \qquad e = \sqrt{(a-b)^2 + \frac{4}{\pi}c^2}$$

$$f = \sqrt[4]{ab - c^2}$$

Equations 3.6 – Objects parameters formulas

We notice from these equations that the only values that need to be stored to calculate all the parameters are $\sum_{\forall pixels} 1$, $\sum_{\forall pixels} x$, $\sum_{\forall pixels} y$, $\sum_{\forall pixels} x^2$, $\sum_{\forall pixels} y^2$, $\sum_{\forall pixels} xy$, $\sum_{\forall pixels} v$ and $\sum_{\forall pixels} v^2$. An application only needs to keep these values in memory and when one of the

object's parameters is required, it can be directly calculated from them. In each of these summations, $\forall pixels$ is intended as "*for all pixels of the object*". When an application wants to calculate these summations, it simply needs to go through all pixels once and update the summations accumulators, as shown in Algorithm 3.2. Note also in the Colour Mean and Colour Variance equations, v is used to represent the value of a pixel. This value can represent a few things, depending on the type of data used and on what we want to calculate. Here are two ways of using the colour mean and colour variance equations:

- v can represent the intensity of a pixel, where the R, G and B values are combined into one. The result of this will be an intensity mean rather than a colour mean.

- We can calculate three colour means, one for each separate value of R, G and B. This has the advantage of differentiating between different colours of same intensity, but requires more calculations.

```
Initialize Area, SumX, SumX2, SumY, SumY2, SumXY, SumV and SumV2 to
        zero
Initialize MaxX and MaxY to zero
Initialize MinX and MinY to the largest number possible

for all the pixels in the object
{
        x = x coordinate of pixel
        y = y coordinate of pixel
        v = colour value of pixel

        if (x > MaxX) then MaxX = x
        if (y > MaxY) then MaxY = y
        if (x < MinX) then MinX = x
        if (y < MinY) then MinY = y

        Area = Area + 1
        SumX = SumX + x
        SumX2 = SumX2 + x*x
        SumY = SumY + y
        SumY2 = SumY2 + y*y
        SumXY = SumXY + x*y
        SumV = SumV + v
        SumV2 = SumV2 + v*v
}
```

Algorithm 3.2 – Calculation of required sums for the parameters of objects

3.5 Summary

We have seen in this chapter that the proposed algorithm, *SmartID* separates the main processing algorithm in two distinct sections, namely the identification of moving objects and the isolation of the backgrounds. The identification of the moving objects can be achieved through the segmentation of moving regions that are identified through

multiple levels of filtering of a comparison between the static background and the most current images. The background image is maintained by partially updating it with newly arrived frames through a weighted blending mechanism combined with a static object detector. The two parts of the algorithm greatly depend on the accuracy of the other one, and since the accuracy of the background generation process normally improves with time, then the overall algorithm's accuracy also improves with time. This process has been discussed in detail in this chapter.

Once the moving objects have been identified, then the next logical step in a higher level algorithm would be to describe the objects that have been found. For that reason, an object model has been discussed in which a statistical analysis of the visual representation of the objects could be performed.

In the following two chapters, the design and implementation of an application that makes use of *SmartID* is covered.

Chapter 4

Design of the *Digital Video Surveillance* application

4.1 Introduction

Chapter 3 described the concepts behind *SmartID* and now we will explain how it can be used in a practical application. DVS is a *Digital Video Surveillance* software application that takes advantage of *SmartID* and was developed to prove *SmartID*'s usefulness in a real world application. This chapter covers the design of DVS.

Before the arrival of video on personal computers, video surveillance was done by physically looking at or recording (on magnetic tapes) the video. The video feed could only be watched on television or on specialized monitors. Nowadays, computer systems and specialized video hardware let you watch video remotely through local area networks or even through low bandwidth channels such as modem connections. People now want to have video surveillance at home, so they can remotely look at what is going on in their house while they are away. Storeowners don't want to be bothered with replacing videotapes in their VCR anymore, they now want digital video. The drawback with videotapes for security is in searching through them for incidents. For example, suppose vandalism had taken place overnight at someone's store. Normally, the process would be to review the videotape until we found the incident on it, which could be a long and tedious task. Now imagine that an active video search could be done automatically: the task would then become very simple.

The purpose of DVS is to facilitate the video surveillance experience from a user's perspective by visually identifying moving objects and by handling the storage and retrieval of digital video efficiently through using *SmartID*.

4.2 DVS requirements

The driving force behind DVS was to demonstrate the usefulness and accuracy of *SmartID* in a realistic environment. Video surveillance was chosen since March

Networks had the hardware and software infrastructure to deliver digital video over networks and also because it was at the time concentrating on security systems. From the original requirement and from this idea on the orientation of the application towards security, a larger set of requirements was outlined. Following is the set of requirements that has been the base of the application design:

1- Since video surveillance is often done from a central location on a site, the application must be able to process (using *SmartID*) and display video from a remote camera.

2- For the case where a video clip would need to be processed offline, the application should also be capable of processing video clips with *SmartID*.

3- It is desirable for a security application to be able to pinpoint the focus of attention on a video monitor. Therefore the application should provide some kind of visual notification for objects that are located and identified by *SmartID*.

4- The processed video should be archived so that it is possible to playback a recorded sequence. An approximate scene reconstruction is suitable, as long as objects of interest and their associated approximate background can be visually represented.

5- On top of the fourth requirement, it should be possible to search previously archived video sequences by their periods of activity (when moving objects appear).

With this set of requirements, we think that the validity and usefulness of *SmartID* will be well represented.

4.3 DVS Design

Now that the requirements have been clearly laid out, the following design implications can be derived.

1- On the application user interface, a display area has to represent video that comes from a remote location. Furthermore, since video needs to be displayed and processed at the same time, then the application has to execute the algorithm in real time.

2- When running in video clip mode, the application needs to perform the same way as it does with remote video.

3- Visual notification can be done by putting a coloured rectangle around the objects that have been found in the remote or clip video.

4- A database of video will have to be used so that clips can be reconstructed from the processed video. Since the reconstruction needs only to be approximate, then this means that the storage process can leave aside some part of the original video sequence.

5- It should be possible to search the database mentioned in the fourth implication.

Figure 4.1 visually encapsulates these design implications.

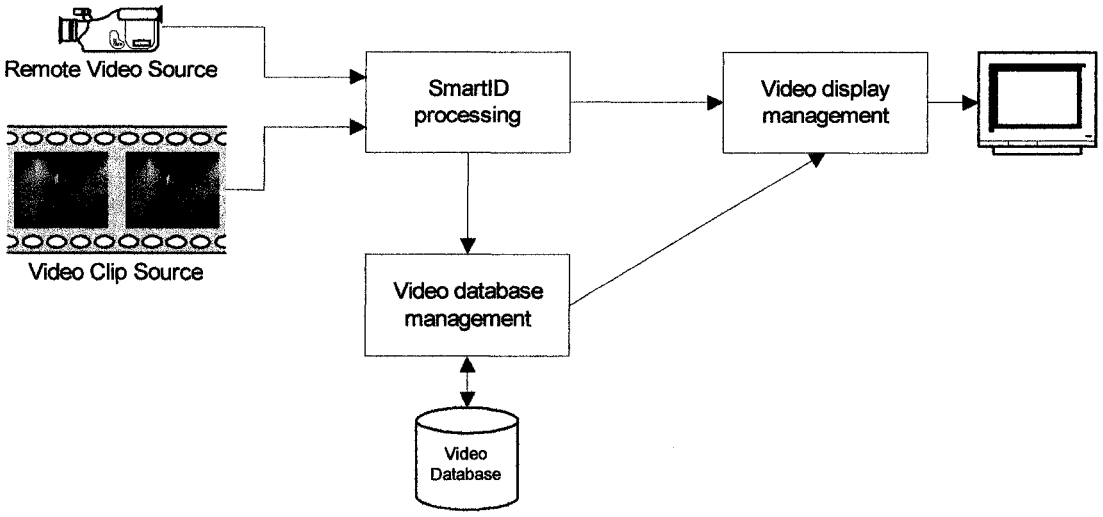


Figure 4.1 – High level design of DVS

We can further specify the components of figure 4.1. These five components are the remote video source, the video clip source, the *SmartID* processor, the video database and the user interface. Following are some more specifications on the design of these items.

4.3.1 Remote video source

Since we are dealing with remote digital video, then the video from the analogue camera will first have to be digitized and then transmitted over some kind of remote connection. Conveniently, March Networks produces what they call video transmission units, or *VTU*s. VTUs can be seen as analogue to compressed digital video capture cards with Ethernet networking capabilities on them. A VTU model 2000 will therefore be used for the digitizing and networking parts for dealing with remote video. Figure 4.2 represents what the remote video source now looks like.

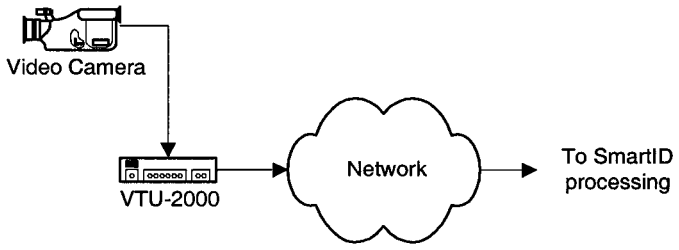


Figure 4.2 – Remote video source

Using a VTU 2000 has some implications on the system and more specifically on the video. This type of VTU encodes video in the H.263 format at different resolutions and frame rates. It has the advantage of producing good quality video at relatively low bandwidth. The video size that is interesting to us here is CIF, which has a frame resolution of 352 pixels by 288 pixels. At this time, the video frame rate cannot be determined since it will depend on the system performance in general. Not only the video will have to be processed by *SmartID*, stored in a database file and displayed, but it

will also have to be decoded before it reaches the *SmartID* processor since *SmartID* works with uncompressed images (see chapter 3). Unfortunately, H263 decoding takes up a considerable amount of CPU cycles, reducing available CPU power availability for *SmartID* processing. A more detailed discussion on this topic is done in section 6.3.

Note also that encoding and decoding the video before it is processed for object identification is not an ideal situation since much noise is consequently mixed with the decoded video that will be used for processing. The source of the noise comes from the discrete cosine transform (DCT) that is performed in the H.263 encoding. Furthermore, March Networks' H.263 decoder first decodes to Y'UV (4:1:1) and then to RGB, thus losing 75% of the colour information. Ideally, processing by *SmartID* should be done at the source of video so that it does not suffer from incorporated noise and from CPU cycles consumed by decompression. Unfortunately, the VTU's DSP processor already had a fair amount of processing to do for H.263 encoding and could not spend cycles on object identification. *SmartID* will be integrated at the VTU level when the new generation of VTU will be developed at March Networks.

4.3.2 Video clip source

Dealing with video clips as opposed to video from a VTU does not make a great difference from the perspective of *SmartID*. The advantage at the application level is that video sequences can be processed offline when needed. What needs to be done at this component level is to be able to read in video frames from a video file. Assuming that

the implementation is done on a Windows platform, most part of the API for video file reading is already available. This will make the implementation of this component a bit easier. As it was the case for the remote video source using VTUs, the video from a video clip is usually already compressed, meaning that noise will be present when *SmartID* will process the video frames. Unfortunately, there is little to be done about that problem since it becomes extremely impractical to store video in an uncompressed format, before it is read and sent to *SmartID*. Figure 4.3 shows the video clip source.

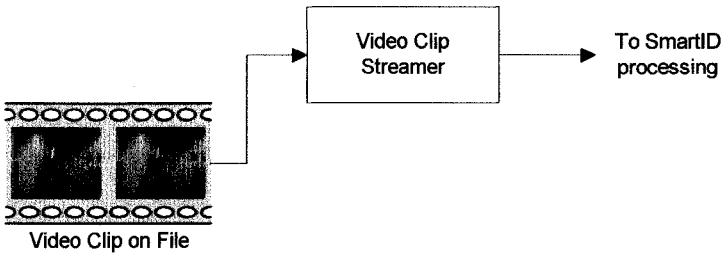


Figure 4.3 – Video Clip Source

4.3.3 *SmartID* processing

Now that we have the means to deliver video to the processing component, we have to think of the format the video will be in when it will be delivered to *SmartID*. We know that VTU 2000s deliver video in H.263 format. On the other hand, video clip formats are not predefined and can be in any format. As chapter 3 explained, *SmartID* works at the visual frame level, and therefore frames must be uncompressed before *SmartID* processing can be done. *SmartID*'s processing unit from figure 4.1 becomes as shown in figure 4.4.

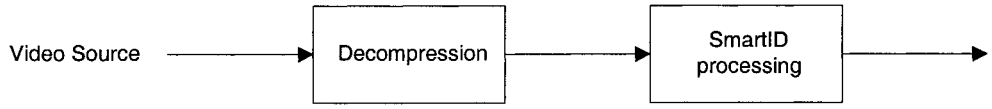


Figure 4.4 – SmartID's processing unit

The decompressor's implementation should be fairly straightforward since it simply is a matter of finding the right CODEC in the system that can decompress the source video. Once the right CODEC has been found, frames are decompressed from it and then sent to the *SmartID* processor.

After *SmartID*'s processing, video information will be comprised of two things: the background approximation image and the objects mask. Since the next components in the system must be able to save, restore and display this information then we need to extract a set of visual objects from the mask of objects and the input frame to *SmartID*. Furthermore we want to simultaneously characterize these objects by a set of parameters as was discussed in section 3.4. This is all shown in figure 4.5.

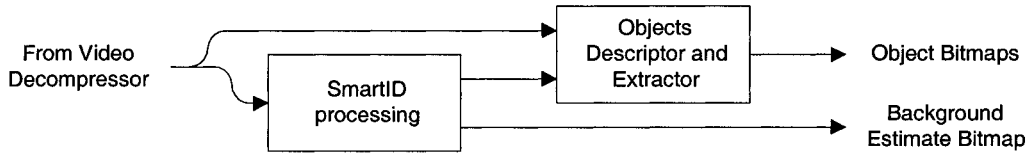


Figure 4.5 – SmartID's processing unit with object description and extraction

At this point, the original video frames can be fully reconstructed (approximately) from the background and objects. As an addition, we will at this point take the

opportunity to demonstrate the usefulness of *SmartID* as a video compressor's aid. Since the background images from frame to frame must be very similar, there may be no point in keeping all of them. We will therefore have a background filter that will let backgrounds go through only when significant changes in them occur. To determine the amount of changes from frame to frame in the background, a cumulative background-to-background thresholded difference can be maintained, and when it has reached some user defined level, a background is let through and the cumulative process restarts. This lossy compression mechanism is more than acceptable for different reasons:

- The valuable information is completely kept in the objects bitmaps.
- Discarded background images are very similar to the previously kept background.
- Frame reconstruction only needs to be approximate.

Now the resulting *SmartID* processing unit from figure 4.1 is shown in figure 4.6.

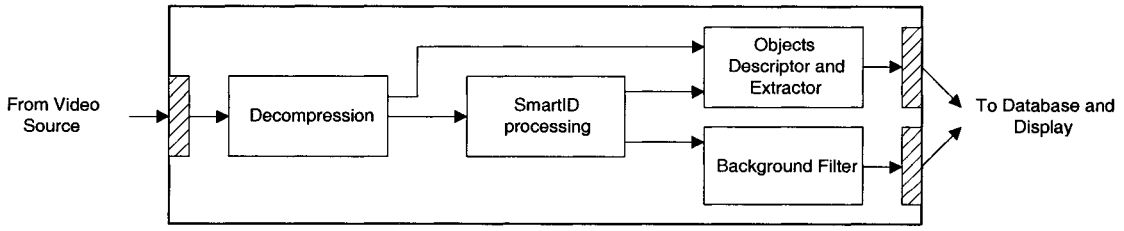


Figure 4.6 – Resulting* SmartID *processing unit

Since the display component obviously does not require the background and object images to be compressed, then they could be sent to it directly. An alternative to that would be to display the original frames from the source instead of a combination of an overlay of backgrounds and object images. The database will also get the uncompressed backgrounds and object images and is discussed in the following section.

4.3.4 Overview of database design

Searching through archived video is normally not something that can be done efficiently, unless someone knows exactly where to look in a video sequence. Video data is usually separated at frame boundaries, leaving no clue as to what is inside the frame. One way of solving this problem is to have someone watching and annotating the video sequence, but this is a very lengthy and inefficient way of doing it. Another solution is to have a computerized mechanism that does this automatically. The latter is more efficient and does not require anyone to be supervising the process, but then we need automatic ways of tagging video with their description. One such way is to use *SmartID*. One interesting feature of it is that frames that contain objects can easily be tagged as such. Searching video for interesting passages would then be a matter of finding object-tagged frames. As it is, *SmartID* as described in chapter 3 only outputs an object mask and background image. At this point objects are all uniquely tagged inside the mask and independent object images (one per object) can be obtained by masking the current frame with each uniquely identified object mask. At the same time, a description of every

object can be done. This is explained in more detail in section 4.3.3. The database that needs to be designed must have ways of quickly searching for objects of interest from an arbitrary point in time. Furthermore, the objects must be joined together with their respective background at the same time such that the original frame will be reconstructed.

In order to retrieve stored information, the database must have an efficient way of looking at the type of stored components (object or background). We have therefore chosen to separate all *SmartID* processed files into an index file and a data file. The index file contains basic information about what is stored in the data file (times, component type, size, position in the file, etc.). On the other hand, the data file contains raw data representing object images and descriptions as well as background images. This has the advantage of preventing searches from being performed on very large video files. In this database scheme, the index file contains the basic information needed to perform simple searches in next to no time. It is extremely small compared to the data file. Figure 4.7 shows the database schema.

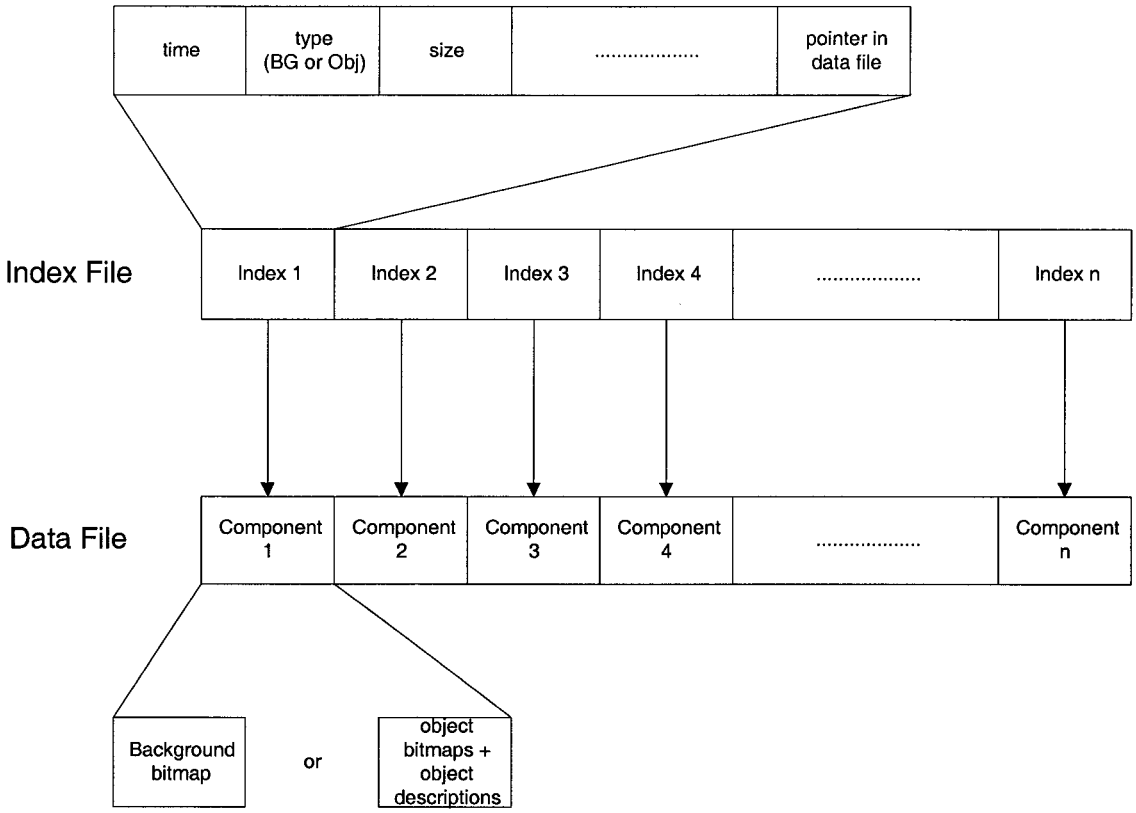


Figure 4.7 – Database schema

In this schema, multiple components can have the same time because through their combination, they make a single entity that represents their respective original frame (i.e. a background can have the same time stamp as an object frame).

Since we know at this point that the data stored in the database is not compressed, then a level of compression will be added to the data file information before storage is performed. The compression algorithm that is used in DVS is discussed in more detail in section 5.3.7. This relates to the implementation level since the algorithm would have to be suited to the format of the video that needs to be compressed.

The compression and decompression that takes place in the database should be transparent to the database users. The storage and retrieval of video from figure 4.1 can then be represented as shown in figure 4.8.

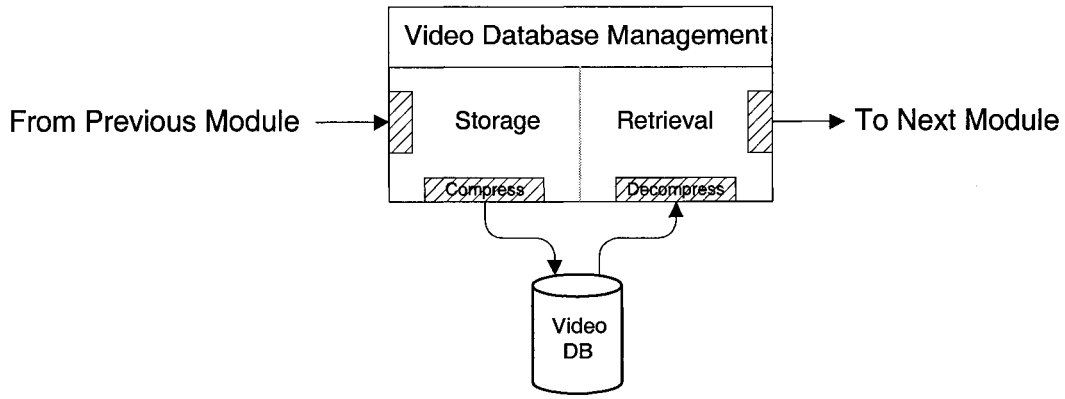


Figure 4.8 – Storage and retrieval of video

4.3.5 User interface

The user interface is what a user would use to control the application. In the case of DVS, the functionality set is quite small since it is primarily built for demonstration purposes. From the set of requirements that are enumerated in section 4.2 and their implications on the application in section 4.3, we can see that there are three modes of operation for DVS. These are the live monitoring mode, the clip monitoring mode and the archive viewer mode. All three modes are accessible through the common main window from which the user decides on the next step to be done for the application. The main window would be the first window that a user sees when launching the application

and should have ways of going to and from the three modes of operation, typically through buttons and menus.

Figure 4.9 shows what the main window looks like.

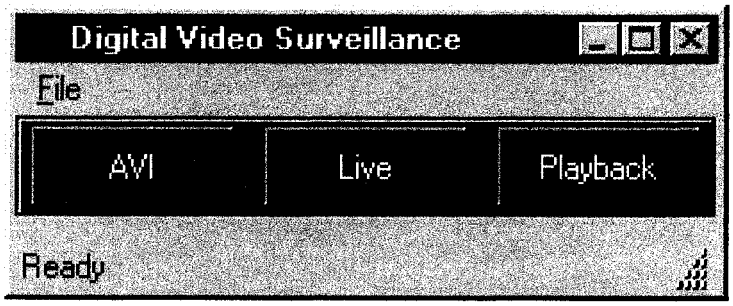


Figure 4.9 – DVS Main Window

In this window, the AVI button and the File / AVI menu selection simply bring the user to the AVI monitoring window after he has selected an AVI source file and an archive destination file. Similarly, the Live button and File / Live menu selection bring the user to the VTU monitoring window, after he has selected the IP address of the VTU video source and an archive destination file. Since the VTU and AVI monitoring modes are very similar, they both are described in section 4.3.5.1. As the third and last function of the main window, the user can get to the archive playback window through the playback button of the File / Playback menu selection. This third mode of operation is described in 4.3.5.2.

4.3.5.1 AVI clip and VTU monitoring

Most of the functionality in both these monitoring modes is the same so they will be treated together. Lets start by looking at both monitoring windows that are shown in figure 4.10.

We immediately notice that both these windows look the same with the exception of the auto repeat button that appears in figure 4.10(a) but not in 4.10(b). Next is a description of every annotated section in both figure 4.10(a) and 4.10(b).

Video display area: This area of the window simply shows the video from the source, enhanced with object rectangles that are sized according to the moving object extents. The object rectangles are coloured to denote distinctiveness between multiple objects in a same frame. At this point, video does not need to be reconstructed from the background and its objects since the original video source is available.

Source close: The source close button can be used to stop monitoring the current video source (VTU or AVI clip) and return to the main window.

Record: The record button indicates if the processed video source is being archived or not. This independent of the fact that the user previously selected a destination archive. This can be useful if a user simply wants to watch the processed video.

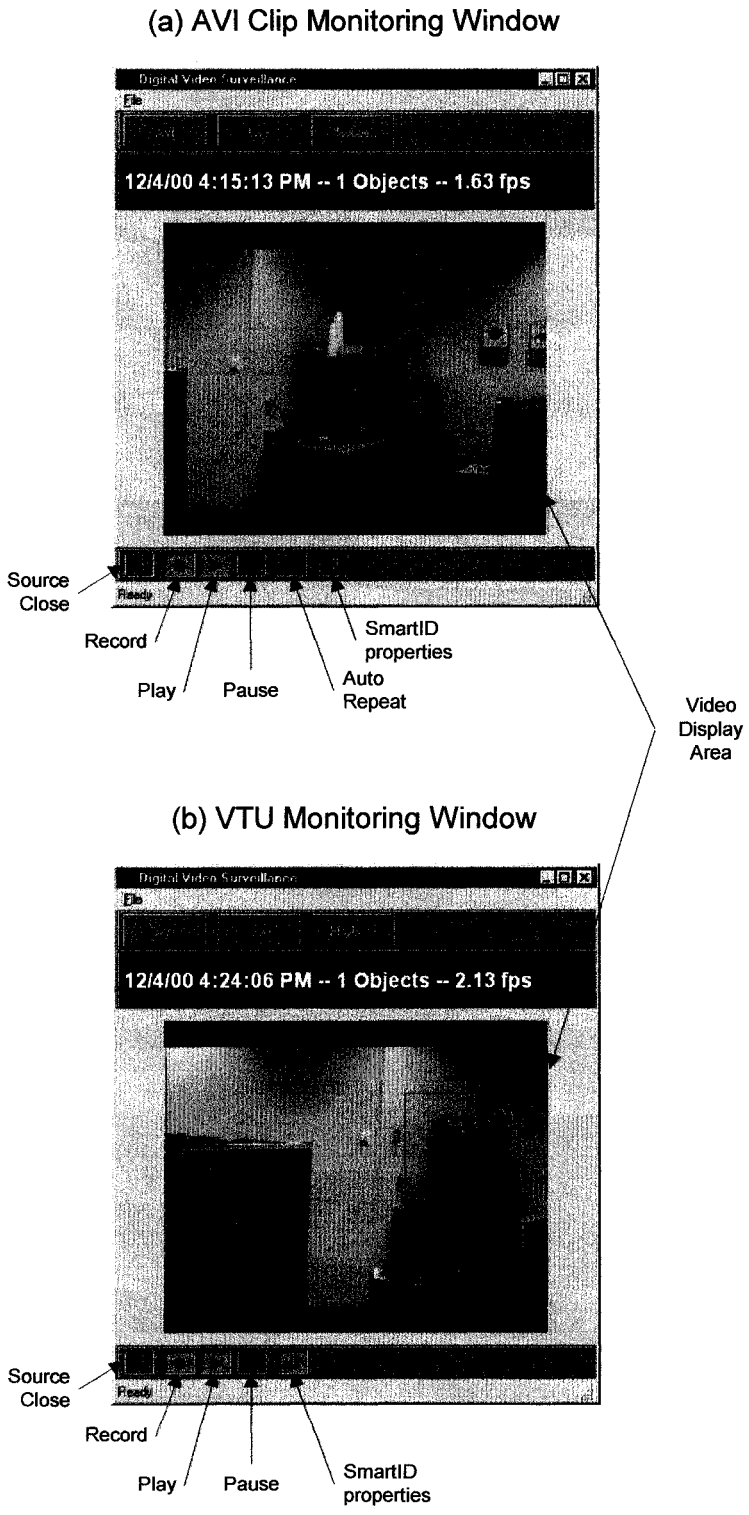


Figure 4.10 – AVI clip monitoring and VTU monitoring

Play: The play button is used to start playing the AVI clip or to start streaming video from the VTU 2000. Once the play button has been pressed, the video source begins being processed by *SmartID*.

Pause: For AVI clips, the pause button stops playing the clip. In the case of a VTU 2000 source, the VTU is told to stop streaming video. When the pause button is down, no video is being processed by *SmartID*. To start processing video again, the start button must be used.

Auto repeat: The auto repeat button is only present when the application is in the AVI clip mode and allows a clip to be played over and over again until the intervention of the user. This button was added for long-term testing of AVI clip sources.

SmartID properties: The *SmartID* properties button is used to bring the *SmartID* properties dialog, letting a user configure all the user-defined parameters of *SmartID*. This dialog is discussed in the following section (4.3.5.1.1.).

4.3.5.1.1 *SmartID* properties dialog

Used in the AVI clip and VTU source modes, the *SmartID* properties dialog lets a user configure every user-definable parameter of *SmartID*. This dialog is shown in figure 4.11.

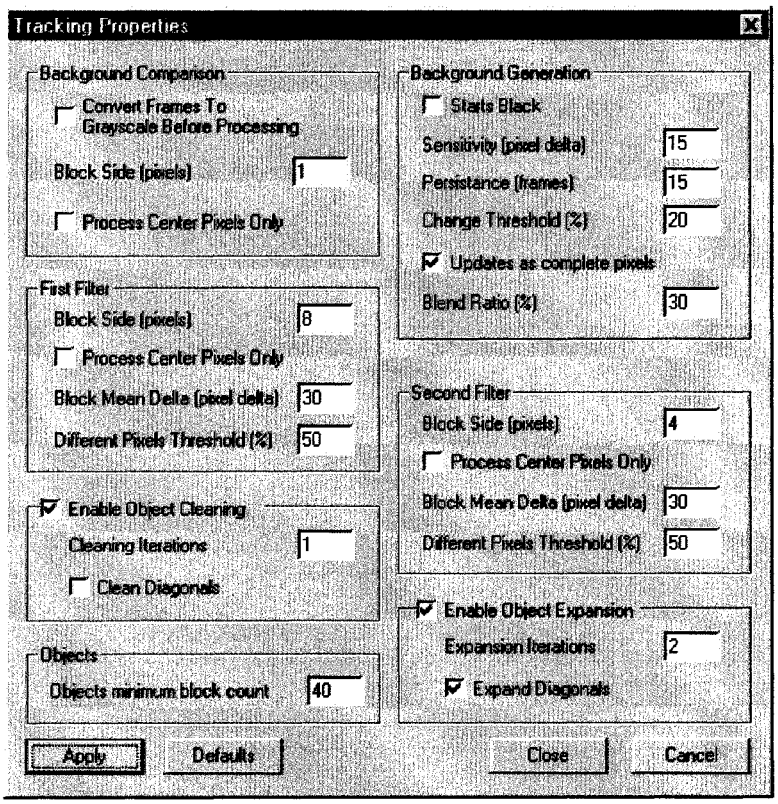


Figure 4.11* – SmartID *parameters

Here is what the available buttons do:

- The apply button applies the changes directly to the algorithm without closing the dialog.

- The defaults button restores default values of all parameters but does not apply the changes.

- The close button applies the changes to *SmartID* and then closes the dialog.

- The cancel button cancels any unsaved changes that were made to the parameters and then closes the dialog.

Next is a list of available parameters accompanied by a brief description of their meaning in the *SmartID*. These are grouped according to their respective sections in the parameters dialog.

Background generation section

- Start black: If this box is checked, then *SmartID* will start the algorithm with a black background image instead of starting with the first frame that it receives from the video. As discussed in chapter 3, this has an effect on the algorithm's adjustment time. When starting black, the background usually does not represent the actual background until it adjusts to it.

- Sensitivity and persistence: These values are used in the static background updates as described in section 3.3.

- Change threshold: The change threshold is a cumulative difference threshold between the generated background frames and is discussed in section 4.3.3 (referred as background filter in that section).

- Updates as complete pixels: In the implementation of *SmartID*, an option to allow background updates to occur at a pixel level rather than at the different layers of the colour spaces was added and this is what this option represents.

- Blend ratio: This is the weight of the new frame compared to older frames for when background updates are done. The blend ratio has been discussed in section 3.3.

Background comparison section

- Convert frames to greyscale before processing: This allows the user to work with greyscale images rather than working with coloured images. Note that the archived video will also be in greyscale if that is selected.

- Block side: This value represents both the height and the width of concerned blocks. Here, the blocks would be used to change the way the differences are made between the frames in order to reduce the CPU cycles needed by *SmartID*. This was not discussed in chapter 3 since it is not an intrinsic part of the algorithm. By default, this value is set to one, so that *SmartID* does a pixel-wise difference between the background and the incoming frames. When this value is set to a value greater than 1, the next property can be used.

- Process centre pixels only: When the block side value is greater than 1 and that this check box is checked, then only one pixel per block is used for the difference

of the frames and the backgrounds. The rest of the differences from the block are assigned the value that results from this single difference. This is used to save CPU cycles while processing the video.

First filter section (for more information on any of these parameters, refer to section 3.2 which describes the filter)

- Block side: This parameter represents both the filter's block height and width in number of pixels. One restriction on this value is that it should be a multiple of the block side of the second filter.

- Process centre pixels only: This parameter allows the algorithm to save some calculation time by not looking at every single pixel in every block. If set, the algorithm will simply look at the block's centre pixel while filtering, but the results of the filtering will be applied to the whole block.

- Block mean delta: The filter is applied on a frame of differences between the background and the incoming frame and the block mean delta parameter represents the threshold of average difference per block for a block to pass the filter.

- Different Pixels Threshold: This parameter represents the percentage of pixels in every block that are not zero. In other words, the percentage of pixels in every block where the background is not the same as the incoming frame.

Second filter section

This section's parameters description is the same as found in the first filter section, with the exception that the block side does not have any restriction (other than being greater than or equal to one. For more information on any of these parameters, refer to section 3.2, which describes the filter.

Enable Object Cleaning section (for more information on any of the following parameters, refer to section 3.2 which describes object cleaning)

- Enable Object Cleaning: Disabling this parameter will prevent the algorithm from performing the border-cleaning filter. This parameter thus enables or disables the whole object cleaning properties section.

- Cleaning iterations: This represents the number of cleaning iterations to be done (if any) before expansion can be performed. We remember from chapter 3 that the cleaning operation involved removing all border blocks from a block mask.

- Clean diagonals: This alters the definition of a neighbouring block as described in chapter 3.2 by telling the cleaning filter to consider or not diagonally connected blocks as being neighbours.

Enable Object Expansion section (for more information on any of the following parameters, refer to section 3.2 which describes object expansion)

- Enable Object Expansion: Disabling this parameter will prevent the algorithm from performing the border expansion process. This parameter thus enables or disables the whole object expansion properties section.

- Expansion iterations: This represents the number of expansion iterations to be done (if any) after cleaning has been performed. Chapter 3 explained how the objects border blocks can be expanded in the objects block mask.

- Expand diagonals: This alters the definition of a neighbouring block as described in chapter 3.2 by telling the expansion process to consider or not diagonally connected blocks as being neighbours.

Objects section

- Objects minimum block count: This represents a threshold for the minimum number of blocks that must be part of an object's block mask before the object is finally stamped as being an object and not just noise.

4.3.5.2 Archive playback

The archive playback window would appear as shown in figure 4.12.

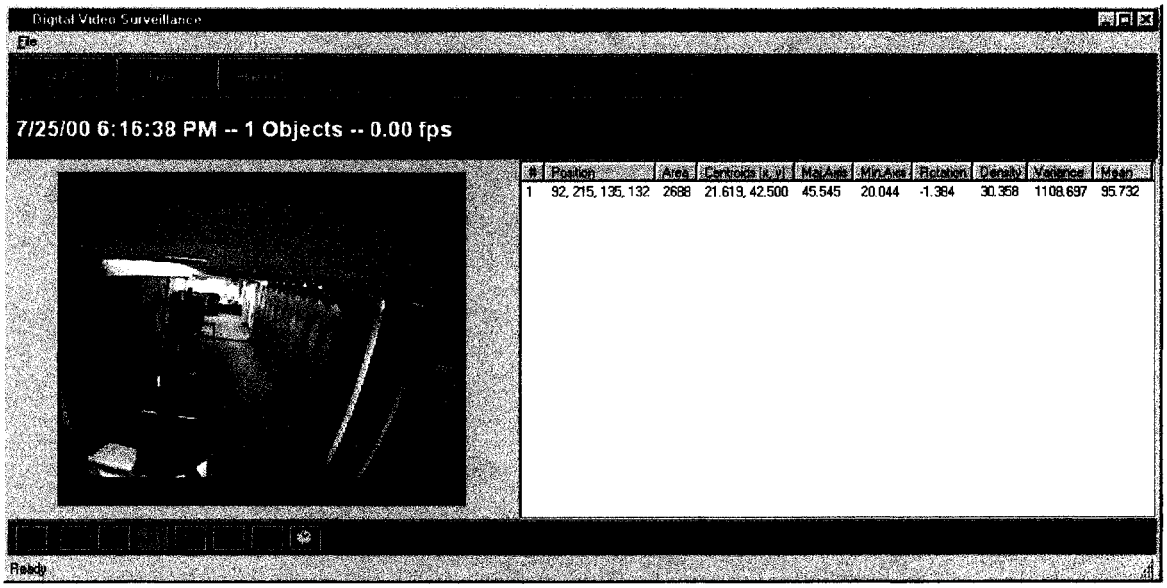


Figure 4.12 – Archive playback window

From this figure, we can see that there are two main areas in the window. The one on the left shows the reconstructed video from the archive, enhanced with object rectangles that are sized according to the moving object extents. Just as in the monitoring video display windows, the object rectangles are coloured to denote distinctiveness between multiple objects that are part of a same frame. The area on the right describes every object that was found during the processing phase. Note here that the description information is simply taken from the archive.

Next is a description of every button that appears in figure 4.12.

Source close: The archive close button can be used to stop viewing the contents of the current archive and returns the user to the main window.

Pause: The pause button simply suspends the viewing of the archive. The other buttons can still be used even if the viewer is paused.

Play: The play button is used to start or resume viewing of the archived video.

Next objects and Previous objects: These buttons give the user the ability to skip the next or previous frames (with respect to the current archive position) that do not contain any object. These represent very well the ability of *SmartID* to facilitate video browsing when looking for interesting sections of an archive.

Set to start and set to end: These buttons are used to set the position of the archive to the beginning or to the end.

Set position: This lets the user position the archive to a time and date that he has to choose. A dialog will pop up when this button is pressed, asking the user to enter a time and date. This pop up dialog is as shown in figure 4.13.

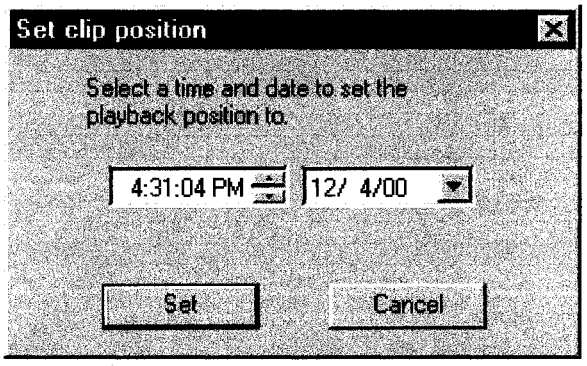


Figure 4.13 – Set position dialog

Speed slider: The speed slider lets a user set the speed at which he / she would like to watch the archived video. This can be useful when the user wants to manually browse though the archived video. Note that this does not skip frames to be displayed but rather reduces the amount of time between displayed frames.

4.4 Summary

We have seen in this chapter that the requirements easily led to the design of the application. The application design has been done in a modular way so that it is easier to replace a component by an equivalent one without having to modify many lines of code. For example, the video source can either be a remote video camera streaming video through a VTU 2000 or can simply be an AVI file. In both cases, the rest of the system can remain the same, making it independent of the video source. The same principle has been applied at a lower level on the video bus in every module. Every higher level module (e.g. processing unit) contains a number of video bus components that are interconnected. From a design point of view, this makes every component simple to design and implement since the problem is seen at a much smaller scale.

Also covered in this chapter is the design of a database that can be used with *SmartID*. It allows independent indexing of backgrounds and objects in order to make quick video searches become a trivial task.

As a final component in the video bus, the user interface has been defined and was kept as simple as it could be, while unleashing the capabilities of *SmartID*.

In the next chapter, the implementation of a digital video surveillance application based on this design is discussed.

Chapter 5

Digital Video Surveillance Implementation

5.1 Introduction

Chapter 4 covered the design of the *Digital Video Surveillance* application, DVS, that is part of this research. The implementation of DVS based on that design is now discussed in this chapter.

The implementation of the core of the application has been done in C++ using Visual C++ 5.0, whereas the application control and user interface has been written in a

programming language called ATCL that is based on C++. ATCL is proprietary to March Networks and is briefly discussed in section 5.2. The whole application has been developed in a Microsoft Windows NT 4.0 environment and requires a VTU 2000 (provided by March Networks) for the streaming of video across a network.

In the following sections, the architecture of the implementation of the video buses as well as the video bus modules themselves are discussed.

5.2 Interfacing with March Networks code base

March Networks has a very stable and large code base, which has been developed over the years. What is interesting is that it already has the ability to stream video from VTUs over IP networks, thus reducing the effort in dealing with this problem in DVS. Furthermore, March Networks has developed a very powerful and easy to use programming language called ATCL that is based on Component Object Model (COM) and C++ (shown in figure 5.1). This provides the ability to write an application's low level modules in C++ and write the higher level structure and control in ATCL. The only extra step that is required is to write a COM interface that exposes the C++ functionality and that will be used in ATCL.

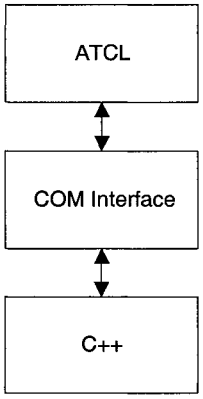


Figure 5.1 – ATCL to C++

Additionally, March Networks already has a very powerful user interface framework that is all exposed to ATCL. This makes it very easy to create things such as toolbars and window layouts.

For the above reasons, DVS is interfacing with March Networks code base and is making use of ATCL for the application's user interface and controls.

5.3 The video bus architecture

When dealing with video-enabled applications, we need a good way of managing video in packet form and also need to have a well-defined video bus architecture. The architecture that is used in DVS is a simple one but at the same time enforces flexibility and code reutilization.

The concept is that video flows in and out of modules that are part of a module chain, where each module has a very specific purpose (e.g. decompress video). Modules

are linked in such a way that they are only aware of the presence of their immediate upstream and downstream modules and know nothing about what these do. This is accomplished by the use of a C++ module base class that has a very limited interface for linking and unlinking modules and also for receiving and sending video packets. The modules of interest would all derive from this base class so that they inherit the module's interface and at the same time add an abstraction layer to the modules functionality. This makes modules see their neighbour modules as black boxes with specific entry points and unknown purpose. This type of module chain is represented by figure 5.2. This figure also shows the flow of video packets in the chain.

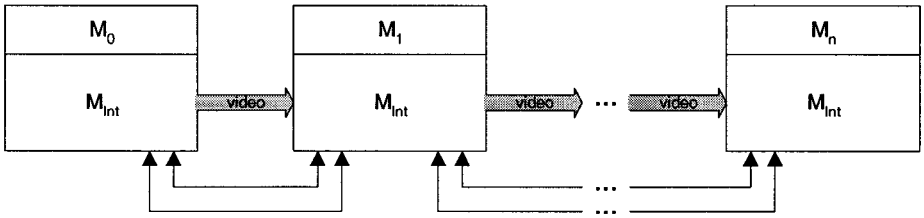
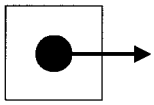
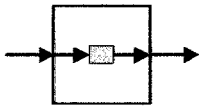


Figure 5.2 – Chaining modules

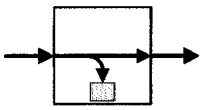
Video modules can generally be classified in one of the following four categories of types. All of these types of modules interact with video packets in their own way.



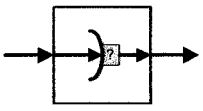
Source modules: This type of module does not have a module that precedes it (an upstream module) and therefore has to produce packets from some video source and then send them to their downstream module if one exists. Example modules of this type are the video clip reader and VTU streamer.



Modifier modules: The modules of this type take their packets from an upstream module, modify or replace them and then send the modified or replaced packets to the downstream module. This type can be viewed as a subset of the source type, with their video source coming from their upstream module. An example of this type of module is the video processing algorithm.



Observer modules: The modules of this type take their packets from an upstream module, process them and then send the original packet to their downstream module. They treat the incoming packets as read-only, but it does not limit them from processing a copy of the video internally, without modifying the source. An example of this type of module is the video display.



Filter modules: Filter modules take their video packets from an upstream module, and then decide if and when they should be sent to their downstream module. An example of this type of module is the packet synchronizer.

As mentioned in section 5.1, the application control and structure is implemented using ATCL, meaning that a COM interface needs to be written in order to expose the required function set and properties of the different modules. Figure 5.2 then becomes as shown in figure 5.3.

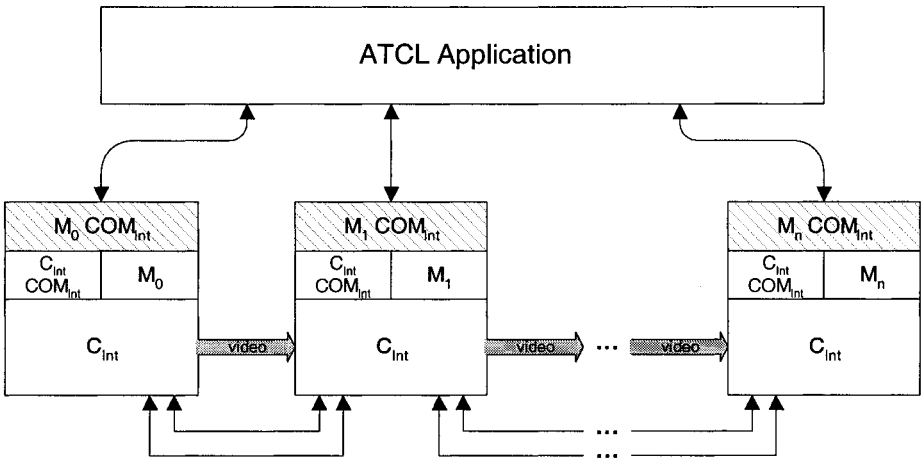


Figure 5.3 – ATCL level module chain

Every COM interface (M_x COM_{int}) therefore exposes the required functionality of the module (M_x) as well as the module's common interface (C_{int} through C_{int} COM_{int}). The ATCL application could then very easily write the bus chain as is done in example code 5.1.

```
object myVideoBus
{
public:
        persist module1 = CreateModule1();
        persist module2 = CreateModule2();
        persist module3 = CreateModule3();
        persist module4 = CreateModule4();

// constructor
        // we want module1 <-> module2 <-> module3 <-> module4
        module1.Link(module2);
        module2.Link(module3);
        module3.Link(module4);
// end of constructor
}
```

Example Code 5.1 – ATCL Video Bus Chain

In this example code, *module1* would likely be a video source module that starts the video packet streaming. It would then have functions such as Open and Close, and StartStreaming and StopStreaming. A button on a user interface could then be calling these functions so that the video starts flowing through this video bus as shown in example code 5.2.

```
object myVideoBus
{
// see example code 5.1
}

// instantiate a video bus
persist m_videoBus = myVideoBus();

function OnStartStreamingButton
{
        m_videoBus.module1.StartStreaming();
}
```

Example Code 5.2 – Accessing module's functionality from ATCL

These code samples show how simple it is to manage video buses through the architecture's flexibility from an application's perspective and also show that modules can be easily reused, as would be the case in example code 5.3.

```
object mySecondVideoBus
{
public:
        persist module1 = CreateModule1();
        persist module2 = CreateModule2();
        persist module3 = CreateModule3();
        persist module4 = CreateModule4();

// constructor
        // now representing module1 <-> module4 <-> module2 <-> module 3
        // note the order of the modules is different from example code 5.1
        module1.Link(module4);
        module4.Link(module2);
        module2.Link(module3);
// end of constructor
}
```

Example Code 5.3 – Demonstration of module reutilization

5.4 DVS modules

Now that section 5.2 explained how the module architecture works, this section will look at the video buses and their modules that are used in DVS.

In figure 4.1 of section 4.3, we could see all the conceptual components of the system, and in sections 4.3.1 through 4.3.4, these components were further decomposed to describe their functionality. Figures 5.4a and 5.4b show how these have all been grouped in modules for the implementation and represent the video buses as they have been implemented. The left side of each module in this figure also shows in which DVS DLL it has been implemented.

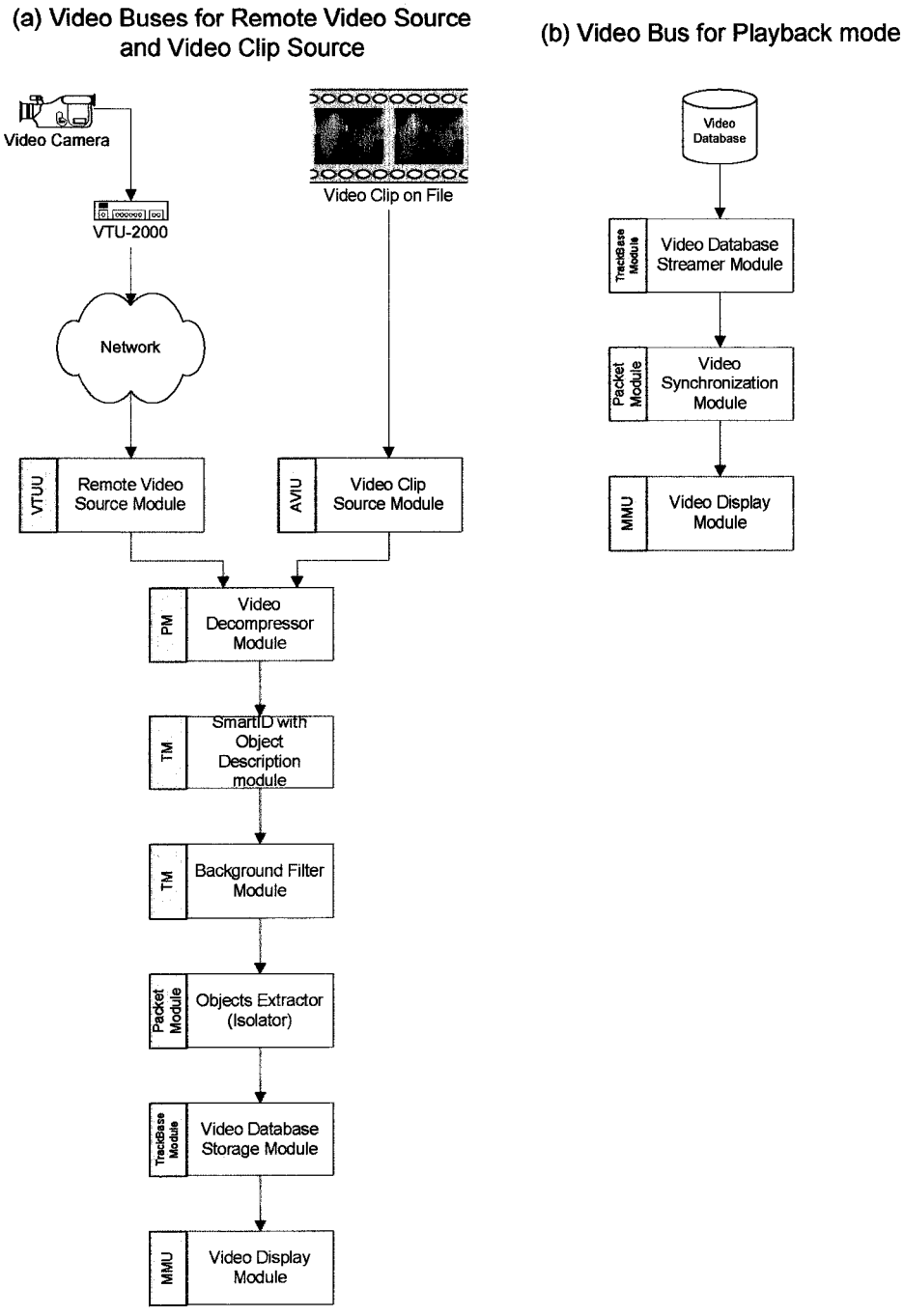


Figure 5.4 – Video buses of DVS

The whole DVS application core is composed of ten Dynamic Link Libraries (DLLs) and a set of files that implement the application's user interface using ATCL. Here are the descriptions of the DLLs:

- General Utilities (*GenU*): Set of low level classes and functions that are used as programming helpers (e.g. critical section class and linked list class).

- Bitmap Utilities (*BmpU*): Set of classes and functions that allow bitmap manipulation and containment such as absolute bitmap difference calculation and system CODEC selector for bitmap decompression.

- Packet Module (*PacketModule*): Provides a set of classes that define packets (e.g. video and timebase) and their metadata. It also can perform video decompression through *BmpU*'s CODEC selector. Lastly, it provides the modules' base class that allows transport of video through module linking.

- Compression Utilities (*CompU*): Set of low level classes providing layered RLE (see section 5.3.7) through an abstract CODEC layer.

- Track Base Module (*TBMod*): Provides a set of database-related classes that allow storage and retrieval of specialized video packets such as the ones produced by *SmartID*.

- AVI Utilities (*AVIU*): Set of classes that allow video packet streaming from AVI video files.

- VTU Utilities (*VTUU*): This DLL acts as a "bridge" between March Networks video packets and video packets used in DVS. The reason why the packet format was redefined in DVS is because March Networks' packets did not allow easy handling of metadata. This DLL therefore greatly depends on March Networks' *VTU.DLL* which allows streaming of video packets from a VTU 2000 source.

- Display Utility (*MMU*): This DLL provides a class that serves as a video display module.

- Identification Module (*TMod*): Provides a set of classes that implement the *SmartID* algorithm.

- Application UI (*GNAUI*): This DLL provides a few dialog templates that are needed in DVS for the user interface.

With the exception of *GenU*, *BmpU* and *CompU* (which are low level DLLs), all other libraries have COM interfaces that externalize their main functionality to ATCL. Figure 5.5 represents the libraries' relationship among themselves.

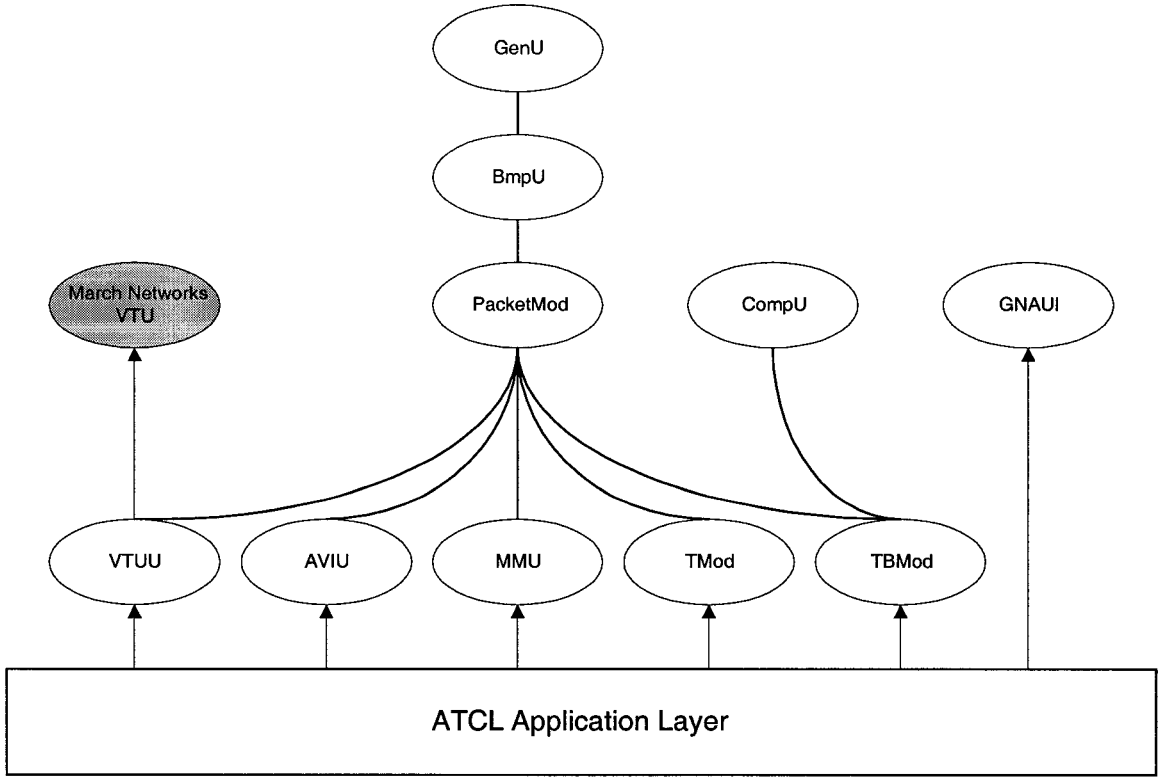


Figure 5.5 – Library dependencies

The following sections give more details on the implementation ideas of the video modules of figure 5.4.

5.4.1 Video clip source module (type "source")

This video module's purpose is to read video data from a specified AVI file, package it in video packets and then send the individual packets to its downstream modules in a synchronized manner. This module does not decompress the video data that it streams, leaving this task to downstream modules.

5.4.2 VTU streaming module (type "source")

The purpose of this module is to take video packets from March Networks' VTU video streaming class and convert it to DVS's packet format so that it can be streamed on DVS's video buses. As it was explained previously, the packets format that March Networks has in place does not currently allow the addition of metadata (extra information) with the packets.

The streaming from the VTU is performed by March Networks' code but this is made transparent to users of the VTU streaming module. The first version of the VTUs that was used in the development of DVS was what March Networks called a VTU model E and was streaming video in FST format (a proprietary compression format of March Networks). Now that this type of VTU is discontinued and unsupported by March Networks, DVS has been modified to work with a new and more powerful VTU called a VTU 2000. This VTU compresses and streams video in a H.263 format. Both VTU models are streaming video over IP local area networks and provide DVS with easy remote video access. VTU-Es are no longer supported by DVS for the reasons mentioned above.

5.4.3 Video decompression for video processing (type "modifier")

The purpose of the video decompression module is to prepare the video in such a way that the video processing module will be able to work with it. At the beginning, a

VTU-E was used, therefore the decompression module had to decompress FST video. March Networks' decompressor only supported decompression to the RGB 32 bits colour space, and therefore the processing algorithm was made to work on RGB data. Now with the VTU 2000, DVS is working with H.263 video. Once again, March Networks CODEC only supported until very recently (October 2000) decompression to RGB. Recent modifications to the CODEC now allow it to decode to Y'UV colour space, from which *SmartID* could obtain some benefits. Some work has been done at March to adapt the implementation of the algorithm to this colour space (see Chapter 7).

Since the video source can really stream any kind of formatted video, the decompressor needs to be generic. For that reason, a direct decompression from H.263 to RGB was not hard-coded. A more general approach was used in which all system installed CODECs are queried for their capability in decompressing the source video into RGB (32 bits). When one such CODEC is found, it is used for decompression of the video stream, which is then ready for processing by *SmartID*.

5.4.4 Video processing module (type "modifier")

This module has the main role of the whole application, which is to identify and describe objects in the video stream. The role of *SmartID* here is only to identify objects in the sequences it is given. The inside of this module therefore appears as shown in figure 5.6.

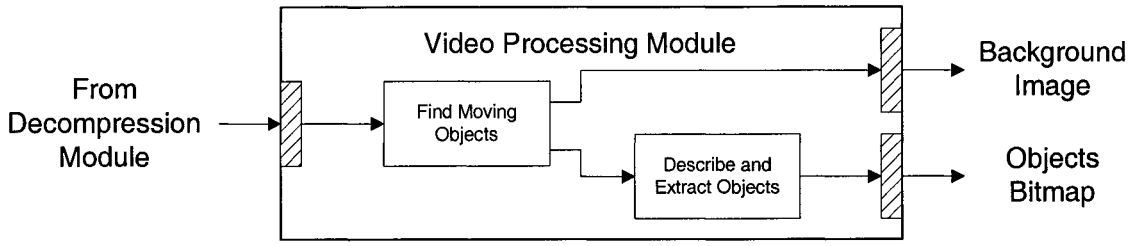


Figure 5.6 – Video processing module

As discussed in section 5.3.3, the incoming video frames need to be decompressed in RGB 32 bits format in this implementation of *SmartID*. The implementation difference would be quite significant if it was performed on Y'UV or $Y'C_BC_R$ encoded video, since these colour spaces don't have the same significance as RGB's colour space. When working with Y'UV or $Y'C_BC_R$, the Y' part represents the "luma", which is a gamma corrected representation of luminance (light intensity). The two other components combined with the Y' give the colour to the images. The Y'UV, $Y'C_BC_R$ and R'G'B' (commonly known as RGB) colour spaces are described by Poynton in more detail in [42] and [43]. The difference with RGB is that the Red, Green and Blue intensities are represented by the R, G and B parts of RGB respectively, thus requiring the algorithm to work on the three colour layers together. This implies that when we mention a "pixel", it refers to all three values (R, G and B) associated to that pixel.

The implementation of *SmartID* must represent the conceptual flow chart of figure 3.11 that deals with object isolation, and also the one from figure 3.16 that deals with the background estimation. Both these conceptual flow charts can be combined together to represent the whole *SmartID* algorithm in its implemented form as shown in figure 5.7. This figure also shows the object description step that is performed after *SmartID*'s processing.

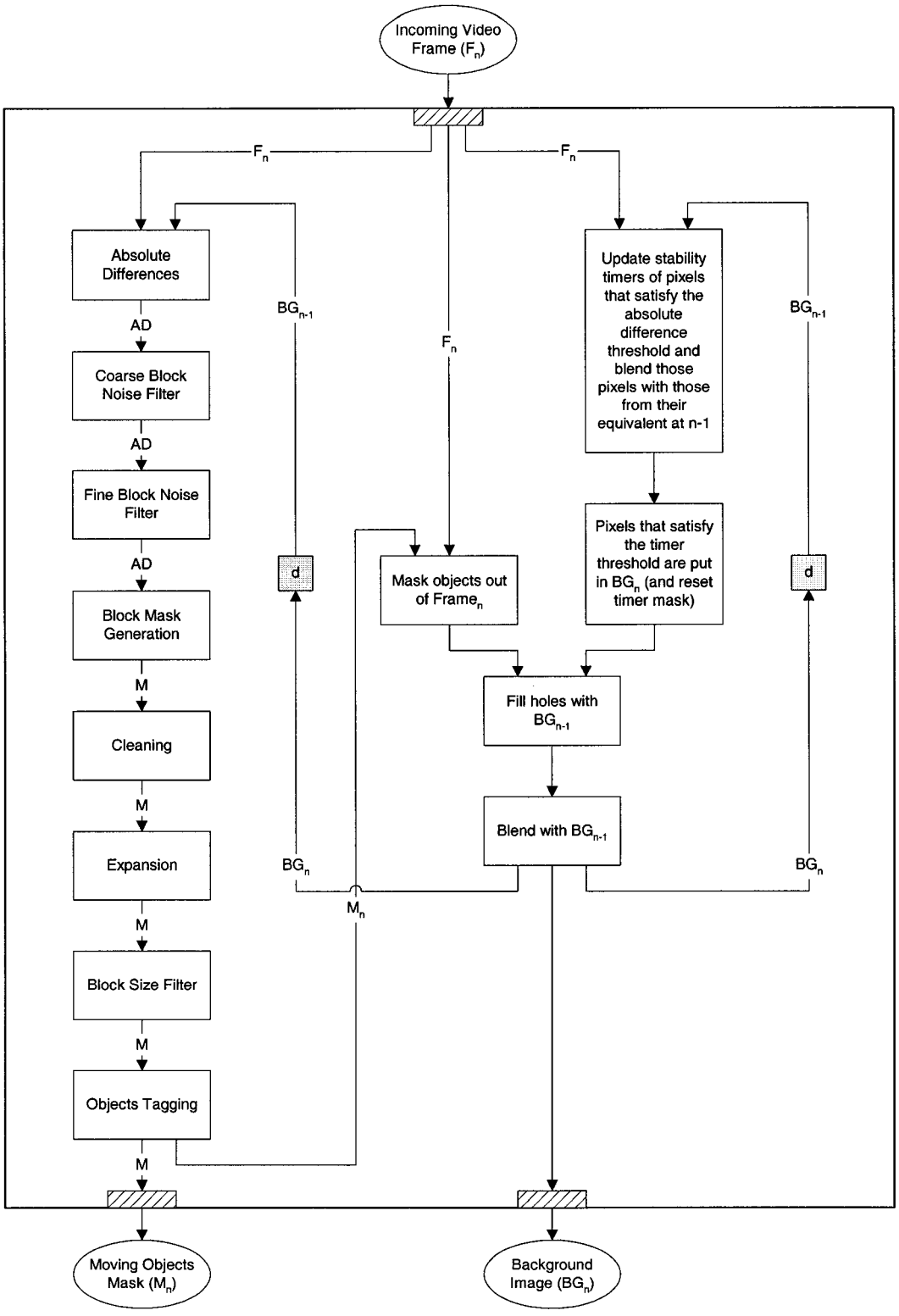


Figure 5.7* – SmartID *conceptual flow chart

Complementary to this figure, we can add that the algorithm was implemented sequentially by first dealing with the objects isolation and then the background estimation.

There are two different types of information that are generated from *SmartID*: an objects mask and a background packet. At this stage, the background packet contains the full background image and a count of pixels that differed from the previous background. As far as objects are concerned, they are all encapsulated in an objects mask that will eventually be combined with the original frame to extract objects information.

The objects descriptor takes the original frame and the objects mask, generates description information for every object in the mask and then packages all that information in a single "objects" packet. This is visually described in figure 5.8.

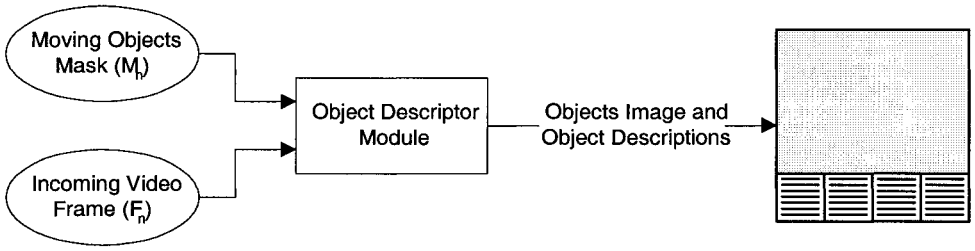


Figure 5.8 – Object descriptor module

At that point, we can let go of the mask and simply rely on the bounding box that is included in every object's description. This will cause some extra visual information not belonging to the objects to be kept with the objects bitmaps, but the descriptive information (apart from position and box size) is still based on the objects pixels only.

The decision of implementing it this way was made because we are dealing with a security system, for which it is important not to lose any information about the objects. In other applications, the mask may need to be kept if the strict boundaries of the object are required.

The descriptive information about the objects includes all values required to calculate the parameters that were discussed in section 3.4. These are $\sum_{\forall pixels} 1$, $\sum_{\forall pixels} x$, $\sum_{\forall pixels} y$, $\sum_{\forall pixels} x^2$, $\sum_{\forall pixels} y^2$, $\sum_{\forall pixels} xy$, $\sum_{\forall pixels} v$ and $\sum_{\forall pixels} v^2$ and bounding box coordinates (minimum x, maximum x, minimum y and maximum y positions). The v values in the last two summations represent in this case the luminance of the pixels as defined by Poynton in [44], which can be calculated from their R, G and B values, as shown in equation 5.1. All the summations can later be used for calculating the colour (luminance) mean, the colour (luminance) variance, the area of the object and the approximating ellipse parameters (x and y centroids, major and minor axis, rotation and density). Section 3.4 described all these parameters and gave the formulas to calculate them.

$$Luminance = 0.212671R + 0.71516 * G + 0.072169B$$

Equation 5.1 – Luminance calculation from R, G and B colour intensities

5.4.5 Background filtering module (type "filter")

This module is a very simple one and is represented in figure 5.9. It accumulates the counts of pixels that are different from background to background (that can be found already in the background packets). It divides the accumulated differences by the total number of pixels in an image and multiplies by 100 (to get the value in cumulative percentage of pixels that were different). When the cumulative difference is lower than a user-defined threshold, it discards the current background since it considers that it is too similar to the previous one. When the threshold is reached, a background packet is let through and the cumulative percentage resets. This is done to reduce transfers of redundant information.

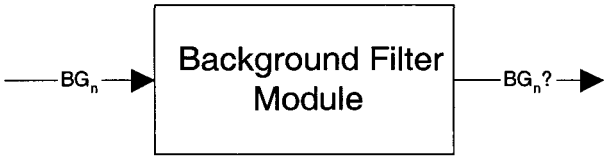


Figure 5.9 – Background filtering module

5.4.6 Objects isolator (type "modifier")

This module has the simple task of repackaging objects packets in a different way than they come when they get out of the video processing module. Figure 5.10 shows the resulting objects packet format. The input frame is composed of a full image and descriptions for a number of objects that are in it. The isolated objects frame contains the boxed objects bitmaps along with their description. The object bitmaps have at this point

been extracted as rectangles with boundaries at the extent of the objects (the boundaries are part of the descriptive information). As mentioned in section 5.3.4, the descriptive information was calculated based on the real objects pixels only, and not on the "boxed" objects pixels, thus leading to more accurate descriptions of the objects. This isolation process is done for different purposes. One of them is to have a smaller size packet in which all the needed information is still fully represented. This is particularly useful for storage.

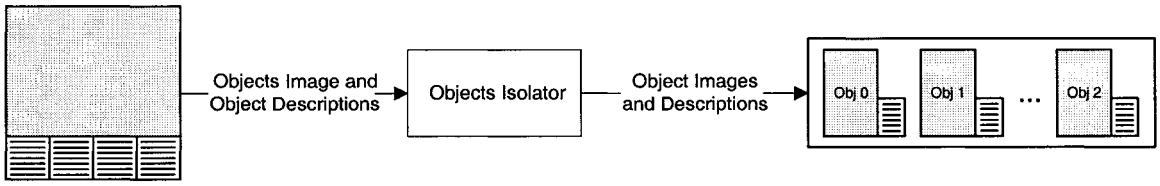


Figure 5.10 – Objects isolator module

5.4.7 Database modules (type "observer")

The database itself accomplished two distinct functions: the storage and the retrieval of video. Since DVS requires only one of these two functions at a time, then two separate modules have been written, one for video packet storage and the other one for video packet retrieval (including search capabilities). Even though the modules have distinct functionality, they are derived from the same database class that does both storage and retrieval. Furthermore, as was discussed in section 4.3.4, compression and decompression of video are also performed at storage and retrieval time respectively. For

more flexibility, the CODEC was made external to the database, so that other CODECs could be used. The database modules are shown in figure 5.11.

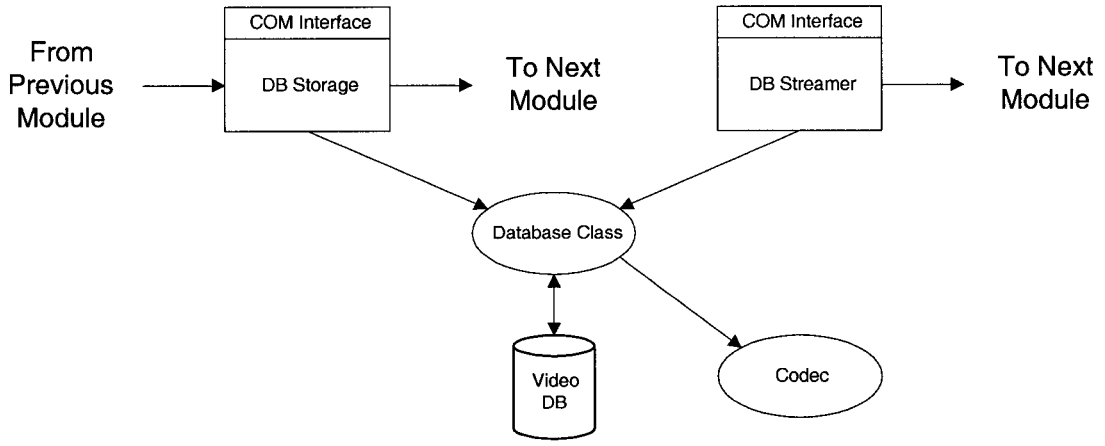


Figure 5.11 – Database modules

The steps involved in the storage of video are as follows:

1- Get video from previous module.

2- Serialize the packet and put it in a buffer.

3- Compress the serialized buffer using an external CODEC (in our case Layered RLE).

4- Store the data in the database.

The steps involved in the retrieval of a video frame are as follows:

1- Get the objects (if any) from the database at the point of interest.

2- From the point of interest, find the latest stored background.

3- Deserialize the objects packet and the background packet to restore the information.

4- Package the background and the objects with their description in a single packet and stream it to the next module in the chain.

An example of the retrieval process is shown in figure 5.12.

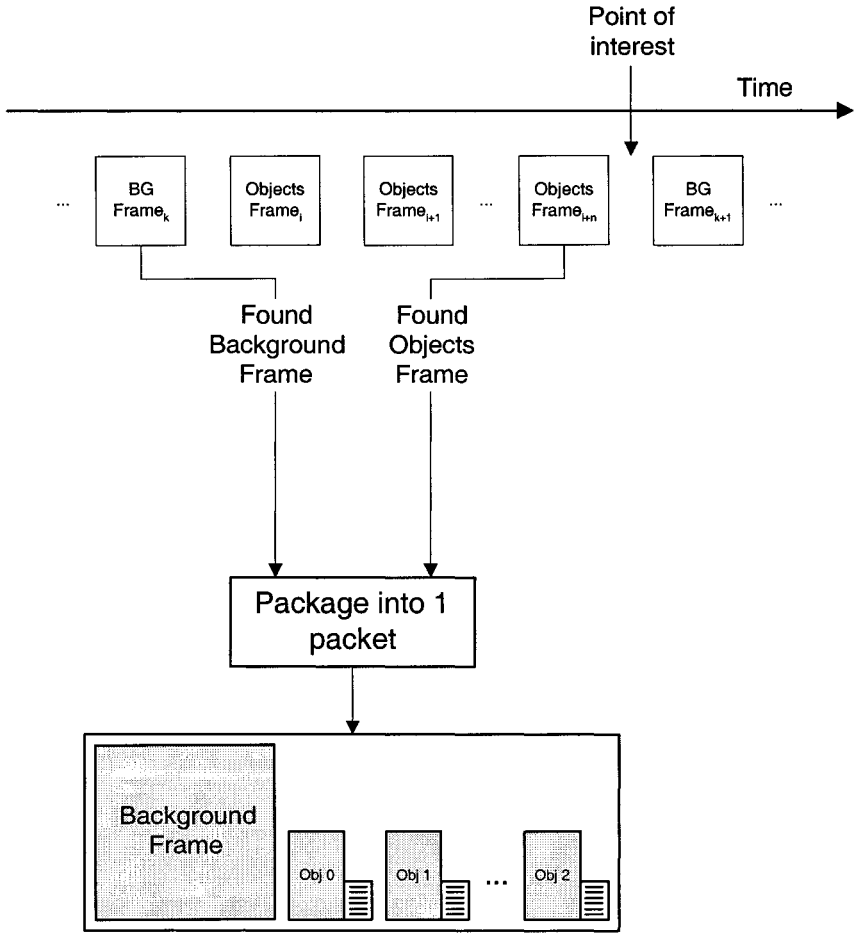


Figure 5.12 – Retrieval process example

As it was discussed in section 5.4.3, the incoming video frames are currently in 32 bits per pixel RGB format. This type of encoding represents each pixel by one Red, one Green and one Blue component. Each of these take 8 bits so there are 8 unused bits left out of the 32. This is done so that pixels are double word aligned, which normally makes processing pixels more efficient because of memory alignment.

Since colours represented by R, G and B are likely to have different values for each of component (i.e. $R \neq G \neq B$), then a normal RLE CODEC would not do a good job at compressing frames formatted this way. This is why we implemented our own CODEC that performs independent compression on RGB layers by reorganizing the source data as shown in figure 5.13.

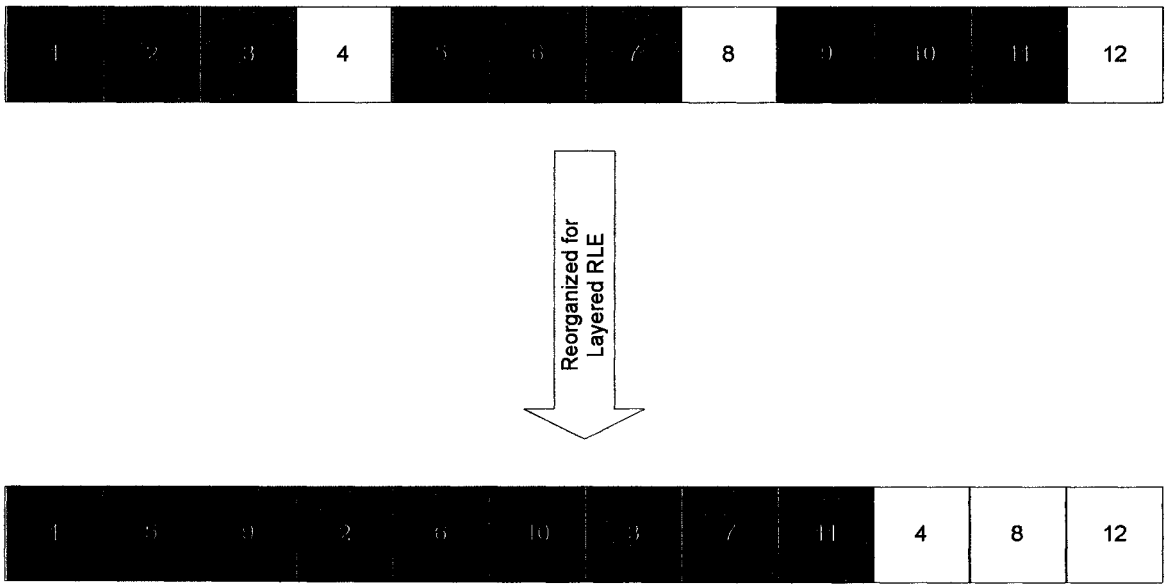


Figure 5.13 – Data reorganization for Layered RLE

When the data is reorganized in this way, the compression will then be faced with consecutive bytes from every layer. Since video normally has very high spatial redundancy, then it is fair to assume that neighbouring values in every place will often have the same value. We can take advantage of this by running an RLE type compression algorithm. Our implementation is as shown in example code 5.4. The decompression is shown in example code 5.5.

5.4.8 Synchronization module (type "filter")

The synchronization module is used only when playing back video that has been processed by DVS. When the database streamer streams the packets on the packet chain, they are not synchronized. Looking at their timestamp, the timestamp of the previous packet, the current time and the time at which the previous packet was sent, we can determine at what time the current packet should be streamed down the chain as show in equation 5.2.

```
Put next input value in buffer A and reset Count to 1

Continue:
Put next input value in buffer B

if (no more input values to come)
{
        if (A == B)
                increment Count by 1

        Output A
        if (Count > 1)
        {
                Output A
                Output (Count - 2)
        }

        if (A != B)
                Output B
}
else if (A == B)
{
        increment Count by 1
        goto Continue
}
else if (Count > 1)
{
        Output A
        Output A
        Output (Count - 2)
        Put B in A
        Reset Count to 1
        goto Continue
}
else
{
        Output A
        Put B in A
        goto Continue
}
```

Example Code 5.4 – RLE compression implementation

```
Start:
Put next input value in buffer A

if (no more input values to read)
{
        Output A
        goto End
}

Continue:
Put next input value in buffer B

if (no more input values to read)
{
        Output A
        Output B
        goto End
}
else if (A == B)
{
        Put next input value in C
        Output A, (C + 2) times
        goto Start
}
else
{
        Output A
        Put B in A
        goto Continue
}

End:
```

Example Code 5.5 – RLE decompression implementation

$TS_n = Relative\ TimeStamp\ of\ packet\ n$
$TS_{n-1} = Relative\ TimeStamp\ of\ packet\ n-1$
$T_n = Send\ time\ of\ packet\ n$
$T_{n-1} = Send\ time\ of\ packet\ n-1$

$$T_n = T_{n-1} + TS_n - TS_{n-1}$$

Equation 5.2 – Calculation of packet send time

At the same time, the synchronization module can serve as a playback speed selector, just by changing the timeline as shown by equation 5.3. In this equation, *PlaySpeed* is the number of times faster than normal speed at which the video should be streamed. The "Speed Slider" in the DVS playback mode uses this special timeline.

$TS_n = Relative\ TimeStamp\ of\ packet\ n$
$TS_{n-1} = Relative\ TimeStamp\ of\ packet\ n-1$
$T_n = Send\ time\ of\ packet\ n$
$T_{n-1} = Send\ time\ of\ packet\ n-1$
$PlaySpeed = Playback\ speed$

$$T_n = T_{n-1} + \left(\frac{TS_n - TS_{n-1}}{PlaySpeed} \right)$$

Equation 5.3 – Calculation of packet send time at arbitrary speed

5.4.9 Video display module (type "observer")

The video display module shown in figure 5.14 simply takes the packets it gets from its upstream module and displays their bitmap on the screen at the right position. It does not need to deal with decompression of the video since in all DVS modes, the video

gets to the video display module in a decompressed form, which can be directly sent to the display.

This module also takes care of putting a coloured box around the objects, according to their description. This is done on the display only, so the video in the packet is unmodified.

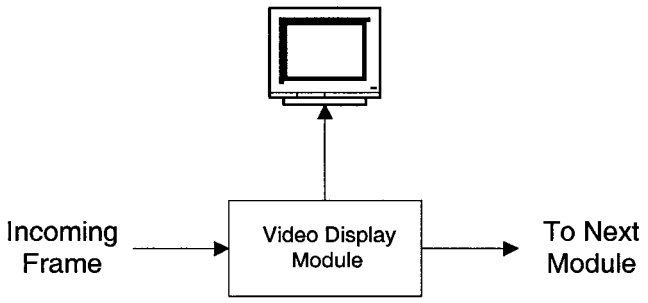


Figure 5.14 – Video display module

5.5 Summary

This chapter covered the implementation details of a digital video surveillance application called DVS that is part of the thesis project. In particular, the video bus architecture and all the modules that are part of the implementation of the three video buses were discussed.

We have seen that the architecture of the video buses has been done in a modular fashion, facilitating the reutilization of code. The implemented modules have in general simple tasks to perform (e.g. decompress video), but their interconnection can lead to

powerful application functionality. These modules are interconnected using March Networks' proprietary programming language called ATCL in a very simple high level manner.

The implementation of *SmartID* has been done in a video processing module that interacts with other modules of the application to enhance their efficiency (e.g. database storage and retrieval). The next chapter evaluates the application performance, in particular the performance of *SmartID*.

Chapter 6

Discussion and test results

6.1 Introduction

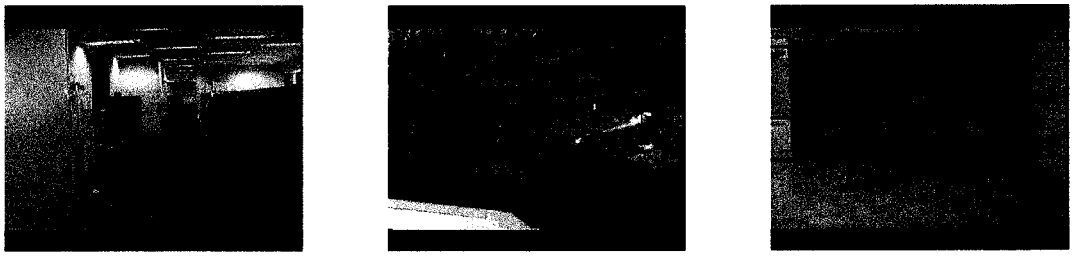
In the previous chapters, it was said that in order to fulfil the requirements of the *Digital Video Surveillance* application, *SmartID* would need to efficiently identify moving objects in digital video sequences. Furthermore, its use would have to help to efficiently store and retrieve the identified objects in a video database. In the following sections, we demonstrate the ability of *SmartID* at achieving the required tasks by presenting some detection results (for efficiency and performance). The use of *SmartID* as a data reduction method for the improvement of video encoders is also presented.

6.2 Efficiency of *SmartID* at detecting moving objects

In order to give an estimation of the efficiency of *SmartID* at detecting moving objects, three pre-recorded video clips in the form of AVI files (in H.263 format) have been used. The first step was to go through the selected video clips frame by frame and manually count the number of moving objects in them. This was done so that we would have a reference count of the interesting objects in clips. Once this task had been performed, we processed each of the video clips with *SmartID* by using the *Digital Video Surveillance* application. Then, by playing back the recorded processed clips, a manual count of the objects that were identified was performed. Both object counts from these observations have been summarized in table 6.1 below. Sample screen shots of each of the video clips are shown in figure 6.1.

	Hallway Close-up	Intersection	Dog
Manually Counted Objects	41	33	19
Counted objects from *SmartID*	42	39	20
Missed objects from *SmartID*	0	0	0
Clip length	60 seconds @ 2 fps	65 seconds @ 2 fps	73 seconds @ 2 fps

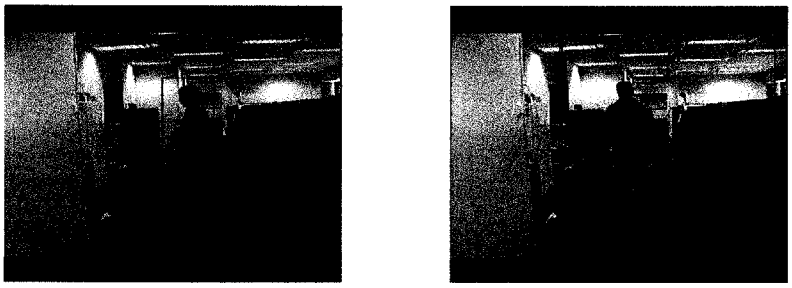
Table 6.1 – Moving objects count for test video clips



(a) Hallway Close-up (b) Intersection (c) Dog

Figure 6.1 – Sample screen shots of video clips being processed

For the Hallway Close-up clip, *SmartID* has detected all the moving objects successfully. The extra object that was detected by *SmartID* was a false detection. This false detection is explained by the automatic gain control of the brightness of the camera. Since the camera was extremely close to the moving objects in the scene (less than a metre), these dramatically affected the brightness of the complete image, thus making the camera adjust its brightness. Because of that, *SmartID* detected a moving object where there was no real motion. It is important to note that when that false detection occurred, a real moving object was present in the scene and was properly detected (although its bounding box was quite large due to the brightness adjustment). If the moving object had not been there in the first place, the false detection would not have occurred (and hence no false alarm). The false detection is not considered critical because of that. Figure 6.2 shows a proper detection and the false detection.



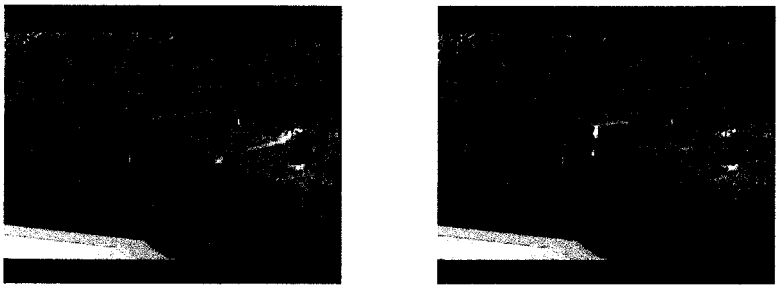
(a) Properly identified object

(b) False detection and enlarged detection

Figure 6.2 – Identified objects in Hallway Close-up clip

In the Intersection video sequence, *SmartID* also properly identified all 33 moving objects. As with the Hallway Close-up clip, *SmartID* made a few extra detections as well. These are all attributed to a very distant shadow from a moving object as is shown by figure 6.3. We can see from that image that the shadow of the passing truck is very distant from the truck itself, making *SmartID* think it was a separate object. Again, since the extra detection of objects have all been triggered by a real moving object in their respective frame, these are therefore not of any real concern. We can say that, since the extra detections do not occur when actual moving objects are not present.

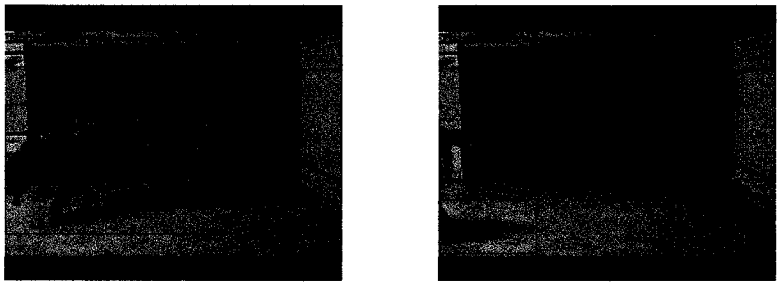
As for the previous two video clips, *SmartID* has properly identified all moving objects in the Dog clip. There were no false detections. The extra object count was simply due to the fact that one of the objects has been detected as two when it was on the scene boundary. The dog's leg and tail appeared as separate objects as shown by figure 6.4 and therefore *SmartID* identified them as two objects. Again, this is not much of a problem here since it would not have caused any false alarm in the system.



(a) Properly identified object

(b) False detection and proper detection

Figure 6.3 – Identified objects in Intersection video clip



(a) Properly identified object

(b) Properly identified object cut in two

Figure 6.4 – Identified objects in Dog video clip

For all three video clips, the results revealed that *SmartID* found all the moving objects manually counted. A false detection occurred in the Hallway Close-up clip, but it turned out not to be of any serious concern because of the context it appeared in (extreme camera brightness adjustment). In the Intersection clip, extra objects were detected. These detections are attributed to the fact that the shadow of a moving truck was very distant and clearly separated from the truck itself. Finally, in the Dog clip there was also an extra object detected. This one was explained by the fact that the dog's leg and tail

were still in the scene (and separated from each other) while the dog itself was not. This does not pose any problem since both these objects needed to be detected in some way.

During the development process, a number of observations have been made concerning the general behaviour of the algorithm. All of these observations comply with the intended behaviour of the algorithm. In all of the scenarios that have been tested, we have never encountered a case where a moving object was not detected. This is exactly what we want the algorithm to do. Furthermore, in all the test cases, newly arrived static objects have eventually been integrated successfully into the static background.

Where the algorithm does not perform so well is with occluded objects (these are normally detected as multiple objects), colliding objects (these are normally seen as single objects), and sudden luminance variations (because the newly arrived video frames are too different from the static background image). It is important to note that SmartID was not meant to address these areas, as they relate to object identification and / or object tracking algorithms.

6.3 Performance evaluation of *SmartID*

The current implementation of the *SmartID* processor does as much processing as it can on the stream it processes and drops any video frame that it did not have time to take care of. This was done so that *SmartID* takes advantage of all the processing power

it can get, meaning that if we run the application on a more powerful computer, we'll get better performance (i.e. frame rate).

The performance evaluation has been done by counting the average number of CPU cycles needed for every frame that *SmartID* needs to process. Although this type of measurement is dependant on a number of factors such as CPU cache memory, pipelining and architecture, it has the advantage of not depending on the CPU speed.

Table 6.2 shows the average number of CPU cycles that were required by *SmartID* while processing four different test clips. They do not take into account the H.263 decompression of the video clips nor the compression and storage time after the frames are processed.

	Hallway	Hallway Close-up	Intersection	Dog
Average cycle count (in million cycles per frame)	156	154	157	151

Table 6.2 – Average CPU cycle count per video frame

The first observation that we can make from these numbers is that for all four video clips, the results are close to one another. And since the four video clips represented different scenes, this leads us to say that *SmartID* is not affected much by the type of images it is processing (which was expected).

To get a feel of the frame rate *SmartID* achieves, we simply have to divide the CPU clock speed by these numbers to get a theoretical average frame rate. For example, running the application on a Pentium II 400 MHz (or 400 million cycles per second), we theoretically get about $400MHz \div 155\frac{million\ cycles}{frame} \approx 2.6\frac{frames}{second}$. In practice, it can process video at a rate of around two frames per second. This is easily explained by the fact that H.263 video decompression time and other tasks that needed to be performed by the application were not taken into consideration in the results from table 6.2. These take a portion of the CPU power that is not negligible in practice.

For the purpose of this project, achieving two frames per second with the Digital Video Surveillance application was satisfying and was enough to prove that the algorithm could be used on live video. Obviously, the performance of the system could be improved in a number of ways. At March Networks, further work has been pursued on the implementation of the algorithm to make it perform better. In the implementation that DVS uses, *SmartID* works on 32 bits per pixel (RGB 32) images and this gets very costly since every pixel has to be visited a number of times for the same calculation. With the enhancements that were performed by March Networks [45], *SmartID* can now process images in the $Y'C_BC_R$ 4:1:1 format which requires only 12 bits per pixel. In the RGB format, all three planes needed to be treated equally and separately, thus tripling the number of calculations. When dealing with $Y'C_BC_R$ formatted images, the C_B and C_R parts only represent the colour space of the image and do not need to be fully processed. This has the advantage of reducing the number of calculations that need to be performed

since most of the calculations done by the modified implementation of *SmartID* are made on the Y' plane (Luma) only. More optimizations have also been made on the image processing routines needed by *SmartID*. These simply deal with the efficiency of the coding and not with the way *SmartID* works. After both these types of optimizations have been performed, the same type of cycle counting has been done and the results are as shown in table 6.3.

	Hallway	Hallway Close-up	Intersection	Dog
Average cycle count (in million cycles per frame)	36	35	36	34

Table 6.3 – Average CPU cycle count per video frame for improved implementation

This proved to be a great improvement over the original implementation. *SmartID* can now achieve a theoretical frame rate of 11.4 frames per second on the same Pentium II 400 MHz and a practical frame rate of about 9.5 frames per second (taking video decompression and other system processes into account). With this kind of performance, we can confidently say that *SmartID* will eventually be able to be integrated in commercial level applications in a near future.

By looking at the results gathered in tables 6.2 and 6.3, we can see that the number of cycles required per video frame is fairly independent from the type of video

that is being processed. This has also been observed in a number of test scenarios during the development process. In fact, the majority of the algorithm's processing power requirements are directly proportional to the size of the images being processed. The small portion that is left of the algorithm depends on the number of objects found in the sequence and also on the size of these objects. This makes the processing power requirements for every frame to be quasi-constant and very predictable.

6.4 Database retrieval results

In general, retrieving specific parts of a large video sequence is not an easy task since it normally requires reviewing the whole sequence manually. One of the advantages of using *SmartID* in DVS (and with the new database design) was that the retrieval process was simplified to a search for the next moving object in the video sequence.

In most cases, finding the first occurrence of a moving object in a video sequence would take as much time as the offset time position of this object from the start of the video sequence. With video that has been processed by the DVS application, the retrieval of this first object is done in just an instant.

As a test case, an overnight video sequence of 10 hours and 35 minutes was used. In that sequence, the first occurrence of a moving object is at 9 hours and 13 minutes from the start of the sequence. Normally, one would have to browse almost the entire

clip to find the first occurrence of the object. Arguably, they could browse through it in fast forward mode at a 16 times speed increase so that they can still see motion from the video. In the case of the overnight clip, it would take about 35 minutes (9h and 13min divided by 16) for a person to find the first occurrence of motion in the 16 times speed conditions. Using DVS to playback and find the first moving object with the same clip took only 164 milliseconds.

We believe that *SmartID* can be an efficient means of improving video database access time when motion needs to be searched.

6.5 Improving the efficiency of video encoders

A compression mechanism was added to DVS at the database storage level just to demonstrate where the data would be compressed after it has been processed by *SmartID*. The compression algorithm that was put in place is a lossless one that doesn't take much processing power. As a result, the compression that is achieved from it is not very good (and was not intended to be), but simple calculus can easily demonstrate the usefulness of *SmartID* as an enhancement to a video encoder. This section demonstrates how the enhancement would be achieved and how well the resulting video encoder would perform. Note that the numbers that are derived in this section are only estimates and represent expected results only.

SmartID identifies the moving objects in video sequences and estimates the background image at every frame. In the implementation of DVS, we have seen that a background filter can be applied after the *SmartID* processing so that only a limited number of backgrounds is kept. This makes sense since the background generally looks the same from frame to frame. This property of the background similarities alone makes way to a great reduction in the bandwidth requirements of the video. We could easily achieve compression ratios of 1 to 100 if only backgrounds were concerned, but unfortunately, the moving objects also have to be taken into account.

We could easily benefit from the use of *SmartID* before the video encoding takes place. Video processing with *SmartID* generates backgrounds (filtered) and objects images that could then be compressed normally through a video encoder. Figure 6.5 shows the modules that could be put in place.

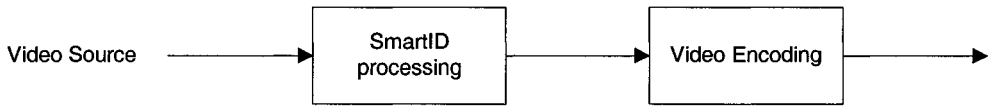


***Figure 6.5 – Modules for enhanced video encoding through* SmartID**

In order to estimate the efficiency of such an enhanced encoder, two H.263 encoded clips have been used for information gathering. The compression estimates that are derived in this section are therefore based on H.263. The information that has been gathered from the original H.263 encoded clips is as shown in table 6.4. The clip named *Hallway* represents an interior scene in a hallway where motion is limited to a few people

walking by from time to time, whereas the clip named *Snow blower* represents an exterior night scene of a driveway with someone snow blowing it, producing extreme "motion" (illumination changes and objects motion) throughout the whole sequence.

	Hallway	Snow blower
Resolution	CIF (352 x 288)	CIF (352 x 288)
H.263 encoded Quality	Highest (from March Networks' H.263 CODEC)	Highest (from March Networks' H.263 CODEC)
Duration	2 minutes and 49 seconds	10 minutes and 4 seconds
Frame count	345 (12 key, 333 inter)	1 228 (41 key, 1 187 inter)
Video data size (no overhead)	1 033 269 bytes (169 453 bytes from key, 863 816 bytes from inter)	11 439 629 bytes (567 740 bytes from key, 10 871 889 bytes from inter)
Average Frame Rate	2.04 FPS	2.03 FPS

Table 6.4 – Test video clips details

We have processed the same two clips with DVS (with a background change level of 20% and a background blend ratio of 20% in both cases) and gathered information on the output of the *SmartID* algorithm on objects and backgrounds (after background filtering). This information is as shown in table 6.5 below.

	Hallway	Snow blower
Number of backgrounds	4	10
Number of objects	101	1 187
Total size (in bytes) of all objects data (not compressed)	1 959 656	277 602 816

Table 6.5 – Test video clips details after processing

From the information in tables 6.1 and 6.2, we can estimate the benefits of processing video with *SmartID* before encoding it with H.263. To do that, we must make sure that we evaluate a worse case scenario with regards to the H.263 compression. We will therefore assume that the H.263 encoding is using key frames only (no temporal relationship between frames), which results in larger frames. In practice, the compression efficiency could be improved by also using interframes, but the goal here is to demonstrate that it is feasible to improve upon an already existing encoder.

We must start by determining the average number of bytes per pixel that are required in the keyframes of the original sequences. This is determined by equations 6.1 and 6.2. These numbers will be used in the approximate compression ratios of the backgrounds and objects of the processed sequences.

$$\frac{169\,453\,bytes}{12\,keyframes} = 14\,121.08\frac{bytes}{keyframe} = \frac{14\,121.08\,bytes}{352x288\,pixels} = 0.1393\frac{bytes}{pixel}$$

Equation 6.1 – Hallway average encoding requirements for key frames

$$\frac{567\,740\,bytes}{41\,keyframes} = 13\,847.3\frac{bytes}{keyframe} = \frac{13\,847.3\,bytes}{352x288\,pixels} = 0.1366\frac{bytes}{pixel}$$

Equation 6.2 – Snow blower average encoding requirements for key frames

We now need to know the number of pixels that need to be stored to represent the whole sequences when they have been processed by DVS. We need to keep in mind that backgrounds are full size frames (352 x 288 pixels) and that the number of bytes per pixel in the uncompressed format for objects is 4 bytes / pixel since the current implementation of *SmartID* uses an RGB 32 format. This is shown in equations 6.3 and 6.4.

$$4\,backgrounds \times 352 \times 288\frac{pixels}{background} + \frac{1\,959\,656\,bytes}{4\frac{bytes}{pixel}} = 2\,365\,160\ pixels$$

Equation 6.3 – Hallway pixel count for objects and backgrounds

$$10\,backgrounds \times 352 \times 288\frac{pixels}{background} + \frac{277\,602\,816\,bytes}{4\frac{bytes}{pixel}} = 70\,414\,464\,pixels$$

Equation 6.4 – Snow blower pixel count for objects and backgrounds

As a final step, we can approximate the number of bytes that would be required to represent the pixels of the two sequences by using the average number of bytes per pixel

in key frames calculated from equations 6.1 and 6.2. Equations 6.5 and 6.6 calculate these values.

$$0.1393\frac{bytes}{pixel}\times 2\,365\,160\ pixels = 329\,453\,bytes$$

Equation 6.5 – Hallway size estimation after H.263 keyframe compression

$$0.1366\frac{bytes}{pixel}\times 70\,414\,464\ pixels = 9\,618\,616\,bytes$$

Equation 6.6 – Snow blower size estimation after H.263 keyframe compression

These values respectively represent 31.9% and 84.1% of the video data size of the original encoded H.263 video sequences. This is of great interest since it demonstrates that *SmartID* can be used as a first step in the compression of video sequences to improve compression efficiency in already existing encoders. According to the results above, the sequences that would be produced by this enhanced H.263 encoder would be from 1.2 to 3.1 times smaller than their equivalent H.263 encoded sequences.

6.6 Summary

In this chapter, we have determined that *SmartID* is very efficient at detecting all kinds of moving objects without ever missing any. We also explained that because there is no tracking and no recognition involved in *SmartID*, it cannot prevent objects from

being detected as two when occlusions occur or when the shadow of objects is seen far away from the objects themselves. This is not seen as failure on the algorithm part because no false alarm would ever be generated from such situations. False alarms would only be generated if objects would be detected in a scene where no objects are present, but *SmartID* never does that.

The performance of *SmartID* in DVS was also evaluated and on average, it can process about two RGB frames per second in that implementation on a Pentium II running at 400MHz. Further improvements on the implementation (not on the design) of the algorithm have been done by myself at March Networks. The performance of the improved implementation has been brought to 9.5 YUV frames per second on the same computer. This demonstrates that the algorithm itself is suitable for real-time applications.

Finally, a theoretical analysis of the impact of *SmartID* as a video encoder was also given. Comparisons between H.263 encoded video and H.263 encoded video enhanced by *SmartID* were made and it was calculated that compression efficiency could be improved by as much as 300%.

Chapter 7

Summary and future work

7.1 Summary

In this thesis, a moving objects identification algorithm called *SmartID* has been designed. A supporting proprietary database and digital video surveillance application (DVS) were also developed to demonstrate the ability of *SmartID* to enhance the storage, retrieval and searching of digital video sequences.

SmartID focuses on identifying moving objects in sequences of video by generating an estimated background image that it compares with incoming video images.

A series of filters is then applied to the comparison frame to determine areas of interest where moving objects are located.

For every frame, the algorithm itself produces a mask of blocks that can be applied to the original frame to get the identified moving objects' images. It also outputs the generated background image that can be used with object images and descriptions in the reproduction of the original video sequence.

The development of DVS proved that *SmartID* is suitable for real-time moving object identification in video sequences in a number of ways. The main characteristics of the application included the storage and efficient retrieval of video, as well as the visual identification of moving objects, all of which were based on *SmartID*'s design. Results have been gathered on the performance and efficiency of the algorithm as implemented in DVS and are as follows:

- *SmartID* can find all moving objects in video sequences,

- On average, 155 million CPU cycles are required to analyze one 352 x 288 video frame, resulting in a two frames per second frame rate on a Pentium II 400 MHz computer. Improvements on the implementation demonstrated that more than 9 frames per second could be achieved using the same computer,

- Retrieval of objects in the video database is extremely efficient.

A theoretical analysis of the benefits of enhancing a video encoder with *SmartID* also demonstrated that the compression efficiency of the video encoder could be improved by as much as 300%.

SmartID can be used by itself for different purposes such as storage, efficient retrieval and reproduction of video sequences, motion detection and video monitoring, but could also serve as a supporting algorithm for other purposes. These include, but are not limited to, object tracking, object recognition, object path determination, video compression and face recognition. Some notes on how *SmartID* could be used in such ways are presented in the following section.

7.2 Future work

As we have mentioned previously, there are possible ways to extend and customize *SmartID* for needs that are more specific. The following paragraphs give an overview of some areas that could be researched.

As it is currently, the algorithm works on images that have been encoded and decoded already. In some cases, *SmartID* could take advantage of the encoded images to determine areas of interest in the stream to limit the motion search areas. One such example is with H.263 encoding which already has motion vectors for blocks of pixels. The disadvantage of such a technique is that it would be tied to a specific CODEC, unless motion vectors are extracted in a generic fashion.

The current implementation of *SmartID* has to deal with considerable amounts of noise coming from the process of compressing and decompressing the video. This noise source could be eliminated completely if the algorithm was ported to a video capture board. In this case, it would work directly with original uncompressed video frames and more accurate detection could be achieved. Furthermore, a specialized hardware platform would most likely improve the rapidity of execution of the algorithm, thus leading to a higher processing frame rate.

For a number of reasons, region specific inclusions and exclusions could be added to the algorithm so that only sections of the frames would be analyzed. This would likely reduce the processing power requirements of the algorithm as well as giving the ability to only identify moving objects in specific sections of the video.

As it was mentioned in section 7.1, *SmartID* can serve as a building block for higher level algorithms such as object tracking. The moving objects identified by *SmartID* would need to be matched with moving objects that have been found in past frames. Furthermore, the history of the objects that are tracked could be kept in order to generate an object path. With a bit more work, object tracking could also solve the problem of colliding moving objects and also the problem of occluded objects.

Since the objects are identified and not recognized by *SmartID*, it cannot determine object types. An object recognition aspect could be added to the algorithm so

that it would detect desired types of objects only. This could be done after *SmartID* has identified the moving objects.

One of the limitations of *SmartID* is that it only works on static cameras. In order to support PTZ cameras, a map of covered 2D space could be built with time so that the background could be kept everywhere the camera would point to. This could be achieved by the use of PTZ camera movement information or by the automatic detection of camera movements through a background matching mechanism.

References

[1] F. Brémond and M. Thonnat, "Tracking multiple non-rigid objects in video sequences", in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, n. 5, pp. 585 – 591, September 1998.

[2] N. B. Hirzalla, "High-Medium level Technical report on SmartFrame, VideoSequenceRecognition, PictorialTranscripts, and Security Applications", Televitesse Systems, Kanata, ON, Canada, Technical Report, May 1997.

[3] N. B. Hirzalla, "An introduction to SmartFrame™", Televitesse Systems, Kanata, ON, Canada, Product Presentation, January 1997.

[4] N. B. Hirzalla, "Media Processing and Retrieval Model for Multimedia Documents Databases", University of Ottawa, Ph.D. thesis, December 1997.

[5] N. B. Hirzalla, "AccesTV SmartFrame™ Overview", Televitesse Systems, Kanata, ON, Canada, Product Overview, February 1997.

[6] J. W. Davis, "Recognizing Movement using Motion Histograms", Massachusetts Institute of Technology Media Laboratory, Cambridge, MA, Technical Report #487, March 1999.

[7] W. Clocksin, "A New Method for Computing Optical Flow", in *Electronic Proceedings of the 11th British Machine Vision Conference*, Online paper: http://www.bmva.ac.uk/bmvc/2000/papers/p13.pdf, September 2000.

[8] B. Galvin, B. McCane, K. Novins, D. Mason, S. Mills, "Recovering Motion Fields: An Evaluation of Eight Optical Flow Algorithms", in *Proceedings of the 9th British Machine Vision Conference*, September 1998, pp. 195 – 204.

[9] S. Yamamoto, Y. Mae, Y. Shirai, J. Miura, "Realtime Multiple Object Tracking Based on Optical Flows", in *Proceedings of IEEE International Conference on Robotics and Automation*, May 1995, vol. 3, pp. 2328 – 2333.

[10] S. Yamamoto, Y. Mae, Y. Shirai, J. Miura, "Optical Flow Based Realtime Object Tracking by Active Vision System", in *Proceedings of 2nd Japan-France Congress on Mechatronics*, 1994, vol. 2, pp. 545 – 548.

[11] Y. Mae, Y. Shirai, J. Miura, Y. Kuno, "Object Tracking in Cluttered Background Based on Optical Flow and Edges", in *Proceedings of 13th International Conference on Pattern Recognition*, 1996, vol. 1, pp. 196 – 200.

[12] B. Bascle, P. Bouthemy, R. Deriche, F. Meyer, "Tracking complex primitives in an image sequence", Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia-Antipolis, France, Research Report 2428, December 1994.

[13] P. Smith, "Edge-based Motion Segmentation", Online research document: http://svr-www.eng.cam.ac.uk/~pas1001/Segment/motion_segmentation.html.

[14] P. Smith, T. Drummond, R. Cipolla, "Edge tracking for motion segmentation and depth ordering", in *Proceedings of the 10th British Machine Vision Conference*, September 1999, vol. 2, pp. 584 – 593.

[15] J.-M. Létang, P. Bouthemy, V. Rebuffel, "Robust motion detection with temporal decomposition and statistical regularization", Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia-Antipolis, France, Research Report 2717, October 1995.

[16] T. Blaszka, R. Deriche, "Recovering and Characterizing Image Features Using An Efficient Model Based Approach", Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia-Antipolis, France, Research Report 2422, November 1994.

[17] A. R. Pope, D. G. Lowe, "Learning Object Recognition Models from Images", in *Proceedings of the 4th International Conference on Computer Vision*, May 1993, pp. 296 – 301.

[18] B. Schiele, J. L. Crowley, "Object Recognition using Multidimensional Receptive Field Histograms", in *Proceedings of the 4th European Conference on Computer Vision*, April 1996, vol. 1, pp. 610 – 619.

[19] D. P. Huttenlocher, W.J. Rucklidge, "A Multi-Resolution Technique for Comparing Images Using the Hausdorff Distance", Department of Computer Science, Cornell University, Ithaca, NY, Technical Report CUCS TR 92 – 1321, December 1992.

[20] D. P. Huttenlocher, G. A. Klanderman, W. J. Rucklidge, "Comparing Images Using the Hausdorff Distance", in *IEEE Transactions Pattern Analysis and Machine Intelligence*, vol. 5, no. 9, pp. 850 – 863, September 1993.

[21] P. C. Hew, "A Stroke Segmentation Cost Function based on the Mahalanobis Distance Classifier", Department of Mathematics, University of Western Australia, Online diary: http://maths.uwa.edu.au/~phew/postgrad/diaries/strsegcostmahal.ps.Z, November 1997.

[22] A. Rangarajan, H. Chui, F. L. Bookstein, "The Softassign Procrustes Matching Algorithm", in *Proceedings of the 15th International Conference on Medical Imaging*, 1997, pp. 29 – 42.

[23] S. Sclaroff, A. Pentland, "Modal Matching for Correspondence and Recognition", in *IEEE Transactions on Pattern Analysis and Machine Intelligence.*, vol. 17, n. 6, pp. 545 – 561, June 1995.

[24] J.-P. Delahaye, "La ressemblance mathématisée", Online document: http://www.pourlascience.com/numeros/pls-235/logique.htm.

[25] S. Edelman, "Computational theories of object recognition", *Trends in Cognitive Sciences*, vol. 1, pp. 296 – 304, 1997.

[26] D. Koller, K. Daniilidis, H.-H. Nagel, "Model-Based Object Tracking in Monocular Image Sequences of Road Traffic Scenes", in *International Journal of Computer Vision*, vol. 10, n. 3, pp. 256 – 281, 1993.

[27] D. Koller, "Moving Object Recognition and Classification based on Recursive Shape Parameter Estimation", in *Proceedings of the 10th Israel Conference on Artificial Intelligence, Computer Vision, and Neural Networks*, December 1993, pp. 359 – 368.

[28] D. Koller, "Model-Based Object Tracking in Road Traffic Scenes", Online document: http://vision.caltech.edu/koller/ModelTracking.html.

[29] G. D. Sullivan, "Visual interpretation of known objects in constrained scenes", *Philosophical Transactions of the Royal Society of London*, series B, vol. 337, pp. 361 – 370, March 1992.

[30] G. D. Sullivan, A. D. Worral, J. M. Ferryman, "Visual Object Recognition Using Deformable Models of Vehicles", in *Proceedings of IEEE Workshop on Context-Based Vision*, June 1995, pp. 75 – 86.

[31] A. D. Worral, G. D. Sullivan, K. D. Baker, "Advances in Model-based Traffic Vision", in *Proceedings of the 4th British Machine Vision Conference*, September 1993, pp. 559 – 568.

[32] D. M. Gavrila, L. S. Davis, "3-D model-based tracking of humans in action: a multi-view approach", in *Proceedings of the 15th IEEE Computer Vision and Pattern Recognition*, June 1996, pp. 73 – 80.

[33] D. M. Gavrila, "Vision-based 3-D Tracking of Humans in Action", Department of Computer Science, University of Maryland, College Park, MD, Ph.D. thesis, 1996.

[34] D. M. Gavrila, V. Philomin, "Real-time object detection for *Smart* Vehicles", in *Proceedings of the 7th IEEE International Conference on Computer Vision*, September 1999, vol. 1, pp. 87 – 93.

[35] M.B. Zaremba, M. Locas, F.A. Gougeon, "La reconnaissance des arbres sur une image multispectrale de haute résolution par des réseaux neuronaux", in *Proceedings of the 16th Canadian Symposium on Remote Sensing*, Sherbrooke, June 1993, pp. 757 – 760.

[36] G. Noël, "The Bidirectional Reflectance Factor of Forest Canopy", Scene Physics and Analysis Section, Major Projects Office, Canada Centre for Remote Sensing, Technical Report, January 1996.

[37] S. M. Smith, "ASSET-2: Real-time motion segmentation and shape tracking", in *Proceedings of the 5th International Conference on Computer Vision*, 1995, pp. 237 – 244.

[38] M. Irani, B. Rousso, S. Peleg, "Computing Occluding and Transparent Motions", in *International Journal of Computer Vision*, vol. 12, n. 1, pp. 5 – 16, 1994.

[39] W. Bell, P. F. Felzenszwalb, D. P. Huttenlocher, "Detection and long term tracking of moving objects in Aerial Video", Online technical report: http://www.cs.cornell.edu/vision/wbell/identtracker/tracksys-s.ps.gz.

[40] J. Stewart, *Calculus – Early Transcendentals*, 2nd ed., J. Hayhurst, Ed. California, USA: Brooks / Cole, 1991, p. 806.

[41] P. J. McKerrow, *Introduction to Robotics*, E. L. Dagless, Ed. Singapore: Addison-Wesley, 1991, pp. 625 – 626.

[42] C. Poynton, "Merging computing with studio video: Converting between R'G'B' and 4:2:2", Online document: http://home.inforamp.net/~poynton/PDFs/Merging_RGB_and_422.pdf.

[43] C. Poynton, "YUV and luminance considered harmful: A plea for precise terminology in video", Online document: http://home.inforamp.net/~poynton/PDFs/YUV_and_luminance_harmful.pdf.

[44] C. Poynton, "Frequently Asked Questions about Color", Online document: http://home.inforamp.net/~poynton/PDFs/ColorFAQ.pdf

[45] G. Noël, "Object Identification", March Networks, Kanata, ON, Canada, Technical Report, November 2000.

Acronyms

ATCL	AccesTV Configuration Language
ATM	Asynchronous Transfer Mode
AVI	Audio Video Interleaved
CIF	Common Interface / Image Format
COM	Component Object Model
CPU	Central Processing Unit
CODEC	COmpressor DECompressor
DCT	Discrete Cosine Transform
DLL	Dynamic Link Library
DSP	Digital Signal Processing / Processor
DVS	Digital Video Surveillance application
FST	Four Squared Transform
H.261	ITU-T video compression standard
H.263	ITU-T video compression standard
IP	Internet Protocol
ISDN	Integrated Standard Digital Network
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
JPEG	Joint Photographic Experts Group
MPEG	Motion Pictures Experts Group
POTS	Plain Old Telephone System
PTZ	Pan Tilt Zoom
RLE	Run Length Encoding
SSD	Sum of Squared Differences
VCR	Video Cassette Recorder
VTU	Video Transmission Unit